Quorus: Efficient, Scalable Threshold ML-DSA Signatures from MPC*

Alexander Bienstock Leo de Castro Daniel Escudero
Antigoni Polychroniadou Akira Takahashi

JPMC AlgoCRYPT CoE & JPMC AI Research
{firstname}.{lastname}@jpmchase.com

Abstract

A threshold signature protocol divides a secret signing key among multiple parties, enabling any subset above a threshold to jointly create a signature. While post-quantum (PQ) threshold signatures are being studied, especially following NIST's call for threshold schemes, most solutions focus on specially designed, threshold-friendly signature schemes. However, real-world applications like distributed certificate authorities and digital currencies require signatures verifiable under existing standardized procedures. With NIST's standardization of PQ signatures and ongoing industry deployment, designing an efficient threshold scheme compatible with NIST-standardized verification remains a critical challenge.

In this work, we present the first efficient and scalable solution for multi-party generation of the module-lattice digital signature algorithm (ML-DSA), one of NIST's PQ signature standards. Our contributions are two-fold. First, we present a variant of the ML-DSA signing algorithm that is amenable to efficient multi-party computation (MPC) and prove that this variant achieves the same security as the original ML-DSA scheme. Second, we present several efficient & scalable MPC protocols to instantiate the threshold signing functionality. Our protocols can produce threshold signatures with as little as 150 KB (per party) of online communication per rejection-sampling round. In addition, we instantiate our protocols in the honest-majority setting, which allows us to avoid any additional public key assumptions.

Our signatures verify under the same ML-DSA implementation for all security levels, with signature and verification key sizes matching ML-DSA; previous lattice-based threshold schemes could not match both of these sizes. Our solution provides the first method for producing threshold post-quantum signatures compatible with NIST-standardized verification, scalable to any number of parties, without new assumptions.

*The full version of this paper may be found at [9].

1 Introduction

In an increasingly interconnected digital world, ensuring the authenticity, integrity, and non-repudiation of electronic communications and transactions is critical. Digital signatures have emerged as a foundational technology in achieving these goals, playing a vital role in securing online communications, enabling trusted software distribution, and underpinning modern cryptographic protocols. By leveraging public key cryptography, digital signatures provide a means to ensure that a given message has not been altered during transmission. Their applications span a wide range of domains, including e-commerce, e-governance, secure email, blockchain, and legal documentation, among others.

Modern digital signatures like ECDSA [45] and EdDSA [8] are vulnerable to attacks given a sufficiently large quantum computer [48]. To address this risk, NIST initiated a series of standardization efforts to construct cryptographic primitives based on quantum-safe assumptions, which are problems that are computationally hard even for quantum computers. This effort resulted in the standardization of a signature scheme based on lattice problems, namely *Dilithium* [4], which was renamed *ML-DSA* [46] in the standard. This scheme is based on the module-lattice learning with errors (MLWE) and short integer solution (MSIS) problems [39], which are believed to be hard even for a quantum machine.

As one of the first standardized post-quantum signature schemes, there is huge motivation to integrate ML-DSA into applications currently using pre-quantum signatures like ECDSA. One core application is *threshold signing*, which delegates the signing authority to n parties. This is a particularly important feature for signing keys that are sensitive and long-lived. Threshold signing balances security and availability by allowing any set of more than t parties to construct a valid signature while preventing any set of t or fewer parties from generating any valid signature. Use-cases of threshold signatures include securing master secrets in key management systems or software update services. In these settings, a secret key can be split across multiple nodes that can still jointly

produce signatures, and yet an adversary would need to compromise at least $t+1$ of these nodes in order to access the secret-key.

Despite many efficient constructions of threshold signature protocols for pre-quantum signatures like ECDSA (e.g., [29]) and Schnorr (e.g., [38, 40]), post-quantum signature schemes tend to be much less "threshold-friendly." In fact, except a few very recent works [11, 30] (see Section 1.2), there does not exist threshold signature scheme that can interoperate with *NIST-standardized* PQ signature verification, such as ML-DSA. Instead, it is common in the literature (cf. [1, 2, 12, 16, 23, 25, 32, 33, 35, 44, 49]) to design lattice-based schemes specifically tailored to the threshold setting, which includes custom verification algorithms that differ from the standardized verification procedures. These signatures have downsides beyond non-standardization: they are typically larger than their non-threshold counterparts, and it is not uncommon that their size and other performance aspects scale with the number of signing parties.

Difficulties in designing threshold ML-DSA. Unfortunately, designing a threshold version of the standardized ML-DSA remains an elusive open problem. Even though in principle one could use generic secure multiparty computation (MPC) to distributively evaluate the ML-DSA signing algorithm, doing so naively would lead to prohibitive costs in practice. This is primarily due to the fact that ML-DSA's native description is highly non-MPC-friendly, since the signing algorithm requires computing a cryptographic hash function (e.g., SHA3) on inputs that must be *hidden* from the adversary. Without additional analysis and modifications to the signing algorithm, this would require running SHA3 within the MPC protocol itself, which is a circuit comprising of millions of arithmetic gates.

1.1 Our Contribution

In this work, we present the *first*[1] threshold signature protocol compatible with the *NIST-standardized* ML-DSA scheme. Our protocol produces signatures that pass exactly the same verification implementation of the standardized, non-threshold ML-DSA scheme. Our protocol supports an arbitrary number of parties, and the distribution of signatures produced by our protocol does not change with the number of parties. One of the main contributions of this work is a new "MPC-friendly" variant of the ML-DSA signing algorithm, where partial signatures can be safely revealed. This allows the expensive hash function in the signing algorithm to be run *locally* by the parties (i.e., outside of the MPC functionality). We provide a security proof for this variant to show that the security level of our variant is essentially identical to the original ML-DSA signing algorithm, and we expect this MPC-friendly variant to find applications in other MPC implementations of ML-DSA (beyond simple threshold access structures). Additionally, we prove that this variant even satisfies a stronger unforgeability notion that we call *unforgeability under chosen message attacks with incomplete signatures* ($\mathsf{UF\text{-}CMA}_{\perp}$), which may be of independent interest.

With an MPC-friendly variant of ML-DSA in place, we present our second contribution, which is a careful synthesis of several MPC building blocks in order to securely evaluate our ML-DSA signing algorithm. The modularity of our approach enables us to leverage state-of-the-art techniques from general-purpose MPC, which allows us to achieve the best performance and trade-offs in terms of communication complexity and round count. In terms of security, we prove that our protocol UC-realizes an ideal functionality, $\mathcal{F}_{\mathsf{T\text{-}MLDSA}}$, which computes ordinary ML-DSA signatures *exactly*, instead of realizing an abstract threshold signature functionality or proving game-based unforgeability in isolation. Furthermore, even though we focus on the setting of honest majority with active security, the generality of our techniques allows for extensions to other contexts as well. When taken together, these contributions yield the first efficient, scalable solution for threshold ML-DSA signature generation.

A conceptual shift away from Schnorr. We emphasize that the use of general-purpose MPC to implement PQ threshold signing represents a major conceptual shift in the design of these protocols. Essentially all prior works on lattice-based threshold signatures based on the Fiat-Shamir transform either rely on threshold homomorphic encryption or attempted to leverage the similarities between Lyubashevsky-like signatures [41] and Schnorr. In threshold Schnorr, it is common for threshold shares of the signature to *themselves* be valid Schnorr signatures under shares of the signing key, and the linearity of Schnorr verification allows these signature shares to be publicly combined. However, locally generating ML-DSA signature shares and running a public, "Schnorr-style" combiner is *fundamentally* limited by the additional requirement on the norm of the ML-DSA signature. In other words, since an ML-DSA signature must have small norm, combining many signature shares increases the norm and degrades the security of the final signature. Therefore, despite the apparent similarities between ML-DSA and Schnorr, our work implicitly argues for a radically different approach in the design of the threshold versions of these signatures. We view this work as identifying the correct MPC subroutines that must be evaluated to obtain a threshold ML-DSA protocol that is simultaneously efficient, secure, and scalable. We expect future works to develop optimized methods for these subroutines in various application settings.

1.2 Related Work

Constructions from Threshold-friendly Schemes. Recent works on threshold lattice-based signatures follow the

[1] As a concurrent and independent work, Borin et al. [11] presented a different approach to threshold ML-DSA. See Section 1.2 for a comparison.

paradigm of threshold Schnorr signatures [6, 20, 38], while simultaneously designing "Schnorr-like" signatures with threshold-friendly properties. Damgård et al. [23] designed a 2-round, n-out-of-n signing protocol for a modified version of the Lyubashevsky signature [41]. This scheme has been improved in several ways, by reducing the number of aborts [3], improving the communication complexity [17], or turning it into t-out-of-n threshold schemes with FHE [35] or MPC [49]. The two-round scheme proposed in [35] obtains signatures of size 46.6 KB and public keys of size 13.6 KB for $(t,n) = (3,5)$. Note that [49] uses generic *dishonest majority* MPC, and like us they rely on the comparison protocol from Rabbit [42]. However, they do not instantiate ML-DSA exactly, and in fact they avoid evaluating hashes in MPC by using commitments as in [23], which is different from our novel approach.

Chairattana-Apirom, Tessaro and Zhu [16] proposed a 2-round threshold scheme based on (a rejection-free variant of) the Lyubashevsky signature with the signature size of [16] is over 100 KB for up to 32 parties [51]. Threshold Raccoon [25] is an efficient 3-round threshold signature scheme based on Raccoon [27], which is itself a threshold-friendly variant of ML-DSA. The scheme naturally supports integration of Shamir secret sharing, allowing efficient instantiations, e.g., signatures of size 12.7 KB and public keys of size 3.8 KB for up to 1024 parties. The follow-up works of threshold Raccoon reduced the number of rounds to 2 [12, 32, 51], proved the adaptive security [36], and realized the security with identifiable aborts [24, 26]. The recent Raccoon-based construction due to del Pino and Niot [44] enjoys the most compact instantiation with signatures of size 2.7 KB and public keys of size 2.6 KB for up to 8 parties by limiting the number of signatures produced to 2^{64}. However, it is unclear whether the same technique carries over to scalable threshold signatures compatible with the standardized ML-DSA verification process.

Two-party ML-DSA signature generation. To the best of our knowledge, Trilithium [30] is the only multi-party scheme outputting a signature compliant with NIST-standardized ML-DSA verification. However, Trilithium is specialized for only two parties, and it requires both parties to generate a valid signature. Moreover, Trilithium heuristically assumes that a rejected, partial signature can be simulated with the uniform distribution, while we introduce an important tweak to the signing algorithm to enable a provable simulation of partial signatures.

Concurrent work. Concurrently with our work, Borin et al. [11] also proposed a threshold signature scheme interoperable with the ML-DSA verification process, optimized for up to 6 parties. Compared to our scheme, their protocol achieves game-based unforgeability in the dishonest-majority setting, requires fewer rounds, and incurs lower communication costs. In contrast, our protocol is proven UC-secure in the honest-majority setting, involves more rounds, and has higher communication overhead for the same number of parties. However, our scheme is scalable to an arbitrary number of parties, whereas their protocol relies on replicated secret sharing and local rejection sampling on individual shares, which seem to limit scalability beyond a small constant number of parties. We compare our protocol to this concurrent work in Section 5.2.

1.3 Technical Overview

In this section we present a brief overview of our techniques. We begin with a high-level description of ML-DSA.[2] This scheme is defined over a standard power-of-two cyclotomic ring $\mathcal{R} = \mathbb{Z}[x]/(x^N+1)$ and its quotient $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$, which means every polynomial of $\mathcal{R}_q$ consists of N coefficients in $\mathbb{Z}_q$. The secret key consists of two vectors $\mathbf{s} \in \mathcal{R}^\ell$ and $\mathbf{e} \in \mathcal{R}^k$ of polynomials, each having small coefficients. The public key is an MLWE instance $\mathbf{t} = \mathbf{A}\cdot\mathbf{s}+\mathbf{e} \bmod q$, where $\mathbf{A} \in \mathcal{R}_q^{\ell\times k}$ is a *public* matrix. For signing a message m, the signer follows the standard *Fiat-Shamir with Aborts* approach [41], summarized as follows:

1. Samples $\mathbf{y} \in \mathcal{R}^\ell$ at random with small entries, and then computes $\mathbf{w} = \mathbf{A}\mathbf{y} \bmod q$. Then (each coefficient of) $\mathbf{w}$ is decomposed as $\mathbf{w} = (2\gamma_2)\cdot\mathbf{w}_1 + \mathbf{w}_0$ for certain public parameter γ_2, where $\mathbf{w}_1$ is *high-bits* of $\mathbf{w}$, while $\mathbf{w}_0$ is *low-bits* of $\mathbf{w}$, respectively.

2. Derive a *challenge* $c = \mathsf{H}(\mathsf{m}, \mathbf{w}_1)$

3. Compute a *response* $\mathbf{z} = c\cdot\mathbf{s}+\mathbf{y}$ (over $\mathcal{R}$).

4. Finally, before outputting the signature, a *rejection sampling* step checks that $\|\mathbf{z}\|_\infty$ and $\|\mathbf{w}_0 - c\cdot\mathbf{e}\|_\infty$ are within certain specified public bounds, aborting by outputting $\perp$ if this is not the case. If the check passes, then the final signature is $(\mathbf{w}_1, c, \mathbf{z})$.

Let us denote by $[\cdot]$ a linear secret-sharing scheme over $\mathcal{R}_q$. A natural approach to thresholdizing ML-DSA would be to let the parties begin with shares of the secret-key $[\mathbf{s}], [\mathbf{e}]$, and run the steps from the signing algorithm, replacing all operations by *generic MPC* calls that operate on shares. The first blocker one would face is evaluating $\mathsf{c} \leftarrow \mathsf{H}(\mathsf{m}, [\mathbf{w}_1])$ in MPC, which is prohibitively expensive due to the complexity of SHA-3 hash function used in ML-DSA. Note that, *for a valid signature*, $\mathbf{w}_1$ is ultimately public since $\mathbf{w}$ can be approximately computed from the signature as $\mathbf{w} \approx \mathbf{A}\mathbf{z} - c\cdot\mathbf{t}$ (and hence $\mathbf{w}_1$, the upper bits, can be *exactly* derived). This leads to the intuition that it is acceptable to let the parties first reconstruct $\mathbf{w}_1 \leftarrow [\mathbf{w}_1]$, and evaluate $c = \mathsf{H}(\mathsf{m}, \mathbf{w}_1)$ *in the clear*.

Security of ML-DSA with Revealed Incomplete Signatures. However, notice that in Step 4 of the (non-threshold) signing algorithm above, the signer outputs nothing *if one of the*

[2] This omits some Dilithium-specific optimizations that are irrelevant to highlight our techniques. We provide a detailed description of actual ML-DSA in Section 2.1 and A.1.

norm checks fails. Is it still secure to reveal $(\mathbf{w}_1, c, \perp)$ when the ML-DSA signature is rejected? We answer this question affirmatively by introducing a tweak to the ML-DSA signing algorithm and go further to prove that the resulting scheme satisfies a stronger version of the standard unforgeability notion.

First, a subtle technicality with rejection-sampling-based signatures [41], including ML-DSA, is that the underlying three-pass identification protocol often lacks a proof of standard *honest verifier zero knowledge* (HVZK). Instead, the standard analysis of ML-DSA [5, 37] relies on weaker variants of HVZK, which only requires the simulatability of the tuple $(\mathbf{w}_1, c, \mathbf{z})$ *conditioned on* $\mathbf{z} \neq \perp$—that is, assuming the transcript has been accepted. While recent works have made progress by showing that rejected transcripts in certain schemes can be simulated statistically [13, 28] or even computationally [28, 44], these techniques do not directly carry over to ML-DSA due to several optimizations and its parameter configurations.[3] Notably, these techniques are developed for identification schemes in which the prover sends $\mathbf{w} = \mathbf{A}\mathbf{y} + \mathbf{e}_w$ as a commitment. They then rely on the leftover hash lemma [13, 28] (viewing $h_{\mathbf{A}} : (\mathbf{y}, \mathbf{e}_w) \mapsto \mathbf{w}$ as a *surjective* universal hash) or the LWE assumption [28, 44] to argue that the rejected $\mathbf{w}$ looks pseudorandom conditioned on $\mathbf{z} = \perp$. However, the ML-DSA identification only sends the higher order bits of $\mathbf{w} = \mathbf{A}\mathbf{y}$ without an error, where the map $h_{\mathbf{A}}$ is not surjective since ML-DSA's $\mathbf{A}$ is either square or tall. To bridge this gap, we tweak the ML-DSA signing operation by bringing back the noise $\mathbf{e}_w$ in such a way that the resulting signature remains verifiable under the standardized ML-DSA verification. The challenging part is to determine a suitable norm bound on $\mathbf{e}_w$ that preserves the standardized ML-DSA parameters while minimizing the correctness error. Through careful analysis, we set $\max \|\mathbf{e}_w\|_\infty = \max \|\mathbf{e}\|_\infty \ll \max \|\mathbf{y}\|_\infty$ to balance these requirements. We empirically confirm that this configuration only introduces a negligible correctness error in practice. As a result, we prove *strong computational HVZK* (scHVZK) [28] for the modified identification under the standard MLWE assumption, without changing the sizes of the output signatures.

With scHVZK in place, it is now straightforward to prove standard unforgeability under chosen-message attacks (UF-CMA) following the analysis of [28]. We also observe that our scHVZK simulator enables us to go beyond standard UF-CMA security. Since a rejected transcript $(\mathbf{w}_1, c, \perp)$ is essentially an *incomplete signature*, revealing such a transcript on queried message m should not count as having signed m. As such, an adversary that only obtains incomplete signatures on some messages should still not be able to forge a valid signature on any of these messages. To capture this intuition, we define the $\mathsf{UF\text{-}CMA}_\perp$ notion tailored to signature schemes that may output incomplete signatures, and prove that our modified ML-DSA satisfies it.

[3] This is a well-known open problem [18]

Our Threshold Signing Protocol. Having removed the challenging part of running H in MPC, what remains is efficiently instantiating the signing steps in MPC. We focus on the setting of *honest majority* with *active security*, for which we use Shamir secret-sharing $[\cdot]_q$ over the ML-DSA modulus q. The parties start with shares $[\mathbf{s}]_q, [\mathbf{e}]_q$ and compute locally $[\mathbf{t}]_q = \mathbf{A} \cdot [\mathbf{s}]_q + [\mathbf{e}]_q$. Most of our more elaborate primitives rely on *edabits* [31]. We use them to sample small shared random values $[\mathbf{y}]_q$ and $[\mathbf{e}_w]_q$, and also to perform the modular reduction to obtain shares of $\mathbf{w}_0$ and $\mathbf{w}_1$, as well as instantiating the secure comparisons needed for rejection sampling. Thanks to our unforgeability analysis sketched above, the parties can safely reconstruct $\mathbf{w}_1$, *even if the signature ends up being rejected*. Because of this, the parties can compute the challenge $c = \mathsf{H}(\mathsf{m}, \mathbf{w}_1)$ *in the clear*. For this rejection sampling part, we need to see if at least one of the bound checks failed, for which we present a method for *Batched OR* computation (see Protocol 3). Assuming rejection sampling passes, the rest of the signature consists of opening $\mathbf{z} = c \cdot \mathbf{s} + \mathbf{y}$. We consider concrete instantiations for these primitives: Escudero et al. [31] for edabits, Rabbit [42] for comparison, Nishide-Ohta [43] for bit decomposition (needed for obtaining $\mathbf{w}_0, \mathbf{w}_1$ from $\mathbf{w}$), and for openings and multiplications we consider both round-optimized (all-to-all openings) and communication-optimized ("king-based") variants (see the full version [9] for more details).

2 Preliminaries

Notations For integers a and b such that $a < b$, $[a, b]$ denotes a set $\{a, a+1, \ldots, b\}$. For an algorithm $\mathcal{A}$, $x \leftarrow \mathcal{A}$ means $\mathcal{A}$ outputs a value x. For a set S, $x \xleftarrow{\$} S$ means a value x is sampled uniformly from S.

2.1 Module Lattices and ML-DSA

In this section, we recall the most basic operations in ML-DSA. As we have already sketched the ML-DSA signing function in Section 1.3, we focus on the description of the parameters and computational assumptions here. For the reader's convenience, Section A.1 provides a self-contained summary of the original ML-DSA scheme and the complete list of parameters (Table 2).

Basic Operations. For a positive integer α and any $x \in \mathbb{Z}$, the centered modular reduction operation $x' := x \bmod^{\pm} \alpha$ maps x to a unique integer $-\lceil \alpha/2 \rceil < x' \leq \lfloor \alpha/2 \rfloor$ such that $x \equiv x' \bmod \alpha$. ML-DSA operates on a power-of-two cyclotomic ring $\mathcal{R} := \mathbb{Z}[x]/(x^N + 1)$ and its quotient $\mathcal{R}_q := \mathcal{R}/q\mathcal{R}$. For a ring element $z = \sum_{i=0}^{N-1} z_i \cdot x^i \in \mathcal{R}_q$, the infinity norm of z is defined as $\|z\|_\infty := \max_i |z_i \bmod^{\pm} q|$. For a module element $\mathbf{z} = (z^{(1)}, \ldots, z^{(\ell)}) \in \mathcal{R}_q^\ell$, its infinity norm is defined

analogously: $\|\mathbf{z}\|_\infty := \max_j \left\|z^{(j)}\right\|_\infty$. The Hamming weight $\mathsf{HW}(\mathbf{z})$ is defined as $||\mathbf{z}||_1 = \sum_j ||z^{(j)}||_1$.

We will write S_η to denote all elements $x \in \mathcal{R}_q$ such that $\|x\|_\infty \leq \eta$. We will write $\tilde{S}_\eta$ to denote the set $\{x \mod^{\pm} 2\eta : x \in \mathcal{R}\}$. Note that $\tilde{S}_\eta$ does not contain any polynomials with $-\eta$ coefficients.

Hash Functions. ML-DSA uses SHAKE-256 H essentially in three ways and maps its output to appropriated codomains: (1) public key digest $tr = \mathsf{H}(\rho, \mathbf{t}_1) \in \{0,1\}^{256}$, (2) message digest $\mu = \mathsf{H}(tr, \mathsf{m}) \in \{0,1\}^{512}$, and (3) Fiat-Shamir challenge $c = \mathsf{H}(\mu, \mathbf{w}_1) \in C = \{c \in \mathcal{R} : ||c||_1 = \tau, ||c||_\infty = 1\}$. As the details of these mappings are not relevant to our protocol, we use the same notation H and refer the reader to [4, §5.3] for more details.

Security Levels & Common Parameters. The ML-DSA FIPS standard [46, Table 1, page 15] lists the various parameters for the scheme. In Table 1, we give the parameters that are unchanged for all security levels of ML-DSA. Note that $q := 2^{23} - (2^{13} - 1)$ just under 2^{23}, and elements in $\mathbb{Z}_q$ fit in 23 bits.

Supporting Functions. ML-DSA relies on the following supporting functions. We assume that if the function takes a module element $\mathbf{x} \in \mathcal{R}_q^k$ as input, it applies the operation coefficient-wise to each polynomial in $\mathbf{x}$. See [46, Algorithm 37]) for the full details:

$\mathbf{A} = \mathsf{ExpandA}(\rho)$: Expands a uniformly random 256-bit seed ρ into a larger matrix $\mathbf{A} \in \mathcal{R}_q^{k \times \ell}$.

$(r_1, r_0) = \mathsf{Power2Round}_q(r, d)$: Decomposes $r \in \mathbb{Z}_q$ into (r_1, r_0) such that $r = 2^d \cdot r_1 + r_0$

$(r_1, r_0) = \mathsf{DecomposeMod}(r, \alpha)$: [19] Decomposes $r \in \mathbb{Z}_q$ into unique (r_1, r_0) such that $r = \alpha \cdot r_1 + r_0$, where $r_1 \in [0, (q-1)/\alpha]$ and $r_0 \in [-\alpha/2 + 1, \alpha/2]$ if $r_1 \neq 0$. It is an MPC-friendly and equivalent variant of $\mathsf{Decompose}$ [46, Algorithm 36], and we use them interchangeably.

$r_1 = \mathsf{HighBits}_q(r, \alpha)$: Extracts the high bits r_1 of r by internally running $\mathsf{DecomposeMod}(r, \alpha)$.

$r_0 = \mathsf{LowBits}_q(r, \alpha)$: Extracts the low bits r_0 of r by internally running $\mathsf{DecomposeMod}(r, \alpha)$.

$h = \mathsf{MakeHint}_q(z, r, \alpha)$: Creates a hint bit h, indicating whether adding z to r changes the high bits of r.

$r_1 = \mathsf{UseHint}_q(h, r, \alpha)$: Uses the hint vector h to obtain the corrected high bits r_1 of r.

Computational Assumption. We recall the standard module LWE assumption, required by the security of ML-DSA. Note that the unforgeability of ML-DSA also relies on (SelfTarget)MSIS, which we omit here.

Definition 1 (Module Learning with Errors (MLWE))**.** *For $q \in \mathbb{N}$ and N a power of two, define $\mathcal{R}_q := \mathbb{Z}_q[x]/(x^N + 1)$. For $k, \ell, \eta \in \mathbb{N}$, let $\mathcal{D}$ be some distribution over $\mathcal{R}_q^\ell$. The decision-MLWE problem $\mathsf{MLWE}_{q,N,k,\ell,\mathcal{D},\eta}$ is the task of distinguishing $(\mathbf{A}, \mathbf{A}\mathbf{y} + \mathbf{e})$ from $(\mathbf{A}, \mathbf{u})$ with non-negligible advantage, where $\mathbf{A}$ is uniform in $\mathcal{R}_q^{k \times \ell}$, $\mathbf{y} \leftarrow \mathcal{D}$ and $\mathbf{e}$ is sampled coefficient-wise uniformly from $[-\eta, \eta]$, and $\mathbf{u}$ is uniform in $\mathcal{R}_q^k$. If $\mathcal{D}$ is a uniform distribution over a set S, we write $\mathsf{MLWE}_{q,N,k,\ell,S,\eta}$.*

Parameter =	Value	Description
q	8380417	Polynomial coefficient modulus
N	256	Degree of the polynomial modulus
d	13	Number of dropped bits from the public key

Table 1: Common parameters of ML-DSA.

2.2 Secure Multiparty Computation

We make use of MPC to design our threshold ML-DSA signing protocol. To define and argue security of our protocol we use the Universal Composability (UC) framework [14] and refer interested readers to the original works for further details. We also write an introductory self-contained description of UC in the full version of the paper [9]. In this model, we define *ideal functionalities* (denoted here by $\mathcal{F}_{\mathsf{name}}$) that may receive inputs from the parties, perform a given computation, and may return some outputs. These functionalities are then instantiated by *protocols* (denoted here by Π_{name}) where the parties communicate with each other without any trusted third party. In words, security is defined by means of *simulation*: a given protocol Π_{name} instantiates a functionality $\mathcal{F}_{\mathsf{name}}$ if the interaction of the adversary in the *real world*, where Π_{name} is executed and the honest parties send messages that depend on their (sensitive) inputs, can be simulated in the *ideal world*, where $\mathcal{F}_{\mathsf{name}}$ runs and all the adversary sees is the output.

Our work is set in the *actively secure* model where an adversary can deviate arbitrarily from the protocol execution and yet security is maintained. We do not write this explicitly in our functionalities, but the adversary can cause an abort at any point of the execution—this is called *security with abort*.[4] We assume a model with n parties among which the adversary controls t, where $n = 2t + 1$. We achieve *computational security*, but we remark this is only for using simple cryptographic primitives such as PRFs, and other than that our protocols are mostly information-theoretic secure, and in particular are simple and concretely efficient.

Remark 1 (On the number of parties)**.** *In threshold signing it is common to have a set of N parties that run DKG, among which a given number $n \ll N$ of them must get together to sign, and the adversary cannot sign even when controlling t of the parties. In this work we obviate this, and assume we already have n parties, all of which will sign a message,*

[4] For the sake of simplicity we also omit other UC-related details in our functionality descriptions, like the use of sid and ssid's.

Parameter	ML-DSA-44	ML-DSA-65	ML-DSA-87	Description
λ	128	192	256	Security level
(k,ℓ)	$(4,4)$	$(6,5)$	$(8,7)$	Public matrix dimensions
η	2	4	2	Coefficient bound of the secret and error
τ	39	49	60	Hamming weight of the challenge c
γ_1	2^{17}	2^{19}	2^{19}	Coefficient range of $\mathbf{y}$.
γ_2	$95232 = \frac{q-1}{88}$	$261888 = \frac{q-1}{32}$	$261888 = \frac{q-1}{32}$	Truncation parameter
ω	80	55	75	Max hamming weight of hint $\mathbf{h}$
$\beta := \eta \cdot \tau$	78	196	120	Bound on $\|\|c\mathbf{s}\|\|_\infty$ and $\|\|c\mathbf{e}\|\|$.
M_{rep}	4.25	5.09	3.85	Expected # of reps in rej. samp. loop (original)

Table 2: ML-DSA parameters that vary with the security level.

and the adversary corrupts t of them where $n = 2t+1$. We emphasize this is merely for the sake of exposition, and we can trivially *accommodate the setting above.*

Remark 2 (On generality)**.** *We describe our functionalities in terms of Shamir secret-sharing, but our core ideas do not rely on any specifics of this scheme, or even the honest majority setting. An alternative take could be to use the so-called* arithmetic black-box *(ABB) model, where all is assumed is a secret-sharing scheme that allows for basic operations such as reconstructions, additions and multiplications (cf. [15]). Such approach would enable different instantiations beyond Shamir secret-sharing and even beyond honest majority. Our core ideas are generic enough to accommodate for this model, and we hope future work in threshold ML-DSA for other settings will benefit from them, but here we chose concreteness and simplicity over generality for the sake of exposition and to better present experimental results.*

2.3 Some Core Functionalities

We consider two finite fields: $\mathbb{F}_q$ where q is the ML-DSA modulus (see Table 1), and $\mathbb{F}_{2^k}$ (degree-k extension of $\mathbb{F}_2$) where $k = \lceil \log n \rceil + 1$.

We use $[\![x]\!]$ to denote Shamir secret-sharing over $\mathbb{F}_q$, and $[\![b]\!]_2$ to denote Shamir secret-sharing over $\mathbb{F}_{2^k}$. For the latter we think of the secret b as being in $\mathbb{F}_2$, and we only use the degree-k extension to get enough interpolation points to define Shamir shares over $\mathbb{F}_2$.

We make use of the following functionalities. These are standard building blocks in MPC and we discuss different instantiations in the full version [9].

- $\mathcal{F}_{\mathsf{Open}}$: receives shares of a secret $[\![x]\!]$ (or $[\![b]\!]_2$). If the shares lie on a degree-t polynomial, reconstructs x (or b) and sends it back to the parties. We either use all-to-all reconstruction (1 round, n^2 communication), or reconstruction through a king [22] (2 rounds, $O(n)$ communication).
- $\mathcal{F}_{\mathsf{Mult}}$: receives shares of two values $[\![x]\!]$ and $[\![y]\!]$ (or $[\![x]\!]_2$ and $[\![y]\!]_2$) and ouputs shares of $[\![x \cdot y]\!]$ (or $[\![x \cdot y]\!]_2$) to the parties. We use triple-based multiplication.
- $\mathcal{F}_{\mathsf{Coin}}[D]$: samples $x \in D$ at random and sends x to all the parties. To instantiate this we call $[\![r]\!] \leftarrow \mathcal{F}_{\mathsf{Rand}}$ and open $r \leftarrow \mathcal{F}_{\mathsf{Open}}([\![r]\!])$. As an optimization, for long coins we sample a seed and use a PRF to expand it.[5]
- $\mathcal{F}_{\mathsf{Rand}}[D]$: samples $r \in D$ and sends shares $[\![r]\!]$ to the parties. We write $\mathcal{F}_{\mathsf{Rand}}$ if $D = \mathbb{F}_q$. Our other type of domain D is sampling bounded values, for which we use edabits.
- $\mathcal{F}_{\mathsf{edabits}}[m]$ [31]: samples $r \in \mathbb{F}_q$, bit-decomposes it as $r = r_1 + r_2 \cdot 2 + \cdots + r_m \cdot 2^{m-1}$, and distributes shares $([\![r]\!]; [\![r_1]\!]_2, \ldots, [\![r_m]\!]_2)$.
- $\mathcal{F}_{\mathsf{dabits}}[k]$ [47]: samples random bits $\mathbf{r} \in \{0,1\}^k$ and distributes shares $([\![\mathbf{r}]\!]; [\![\mathbf{r}]\!]_2)$; i.e., edabits of length-*one*.

3 MPC-friendly ML-DSA Signing

In this section, we present our MPC-friendly variant of ML-DSA in Algorithm 1. The purpose of this section is to prove the extended unforgeability of Algorithm 1 (Theorem 1) and thus justify the threshold ML-DSA functionality (see full version [9] for details) which computes Gen and Sign of Algorithm 1 inside the ideal functionality. We then present protocols that UC-realize the threshold ML-DSA functionality in Section 4. As mentioned in Section 1.3, a short error vector $\mathbf{e}_w$ with coefficients in $[-\eta, \eta]$ is added to $\mathbf{Ay}$ before decomposition occurs. The description also simultaneously defines the underlying identification scheme $\mathsf{ID} = (\mathsf{Gen}, \mathsf{P}, \mathsf{V})$, in which the prover algorithm consists of $\mathsf{P} = (\mathsf{P}_1, \mathsf{P}_2)$, and the verifier V returns a uniformly sampled $c \in C$ and then accepts if and only if $\mathbf{w}_1 = \mathsf{UseHint}_q(\mathbf{h}, \mathbf{Az} - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2)$, $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ and $\mathsf{HW}(\mathbf{h}) \le \omega$.

[5] This does not *instantiate* $\mathcal{F}_{\mathsf{Rand}}$ technically, but it suffices for our purposes. See the Full Version [9].

Algorithm 1: MPC-friendly ML-DSA (changes from original ML-DSA are highlighted)

$\underline{\mathsf{Gen}()}$

1: $\rho \xleftarrow{\$} \{0,1\}^{256}$
2: $\mathbf{A} := \mathsf{ExpandA}(\rho)$
3: $(\mathbf{s},\mathbf{e}) \xleftarrow{\$} S_\eta^\ell \times S_\eta^k$
4: $\mathbf{t} := \mathbf{A}\mathbf{s}+\mathbf{e} \bmod q$
5: $(\mathbf{t}_1,\mathbf{t}_0) := \mathsf{Power2Round}_q(\mathbf{t},d)$
6: $tr := \mathsf{H}(\rho,\mathbf{t}_1)$
7: $\mathsf{pk} := (\rho,\mathbf{t}_1)$
8: $\mathsf{sk} := (\rho,tr,\mathbf{s},\mathbf{e},\mathbf{t}_1,\mathbf{t}_0)$
9: **return** $(\mathsf{pk},\mathsf{sk})$

$\underline{\mathsf{Sign}(\mathsf{sk}=(\rho,tr,\mathbf{s},\mathbf{e},\mathbf{t}_1,\mathbf{t}_0),\mathsf{m})}$

1: $(\mathbf{w}_1,\mathsf{st}) \leftarrow \mathsf{P}_1(\mathsf{sk})$
2: $\mu := \mathsf{H}(tr,\mathsf{m})$; $c = \mathsf{H}(\mu,\mathbf{w}_1)$
3: $(\mathbf{z},\mathbf{h}) \leftarrow \mathsf{P}_2(\mathsf{st},c)$
4: **return** $(\sigma=(c,\mathbf{z},\mathbf{h}),\mathbf{w}_1)$

$\underline{\mathsf{Ver}(\mathsf{pk}=(\rho,\mathbf{t}_1),\mathsf{m},\sigma=(c,\mathbf{z},\mathbf{h}))}$

1: $\mathbf{A} := \mathsf{ExpandA}(\rho)$
2: $\mu := \mathsf{H}(\mathsf{H}(\rho,\mathbf{t}_1),\mathsf{m})$
3: $\mathbf{w}' := \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d \bmod q$
4: $\mathbf{w}_1' := \mathsf{UseHint}_q(\mathbf{h},\mathbf{w}',2\gamma_2)$
5: **assert**:
$$c = \mathsf{H}(\mu,\mathbf{w}_1') \wedge \|\mathbf{z}\|_\infty < \gamma_1-\beta \wedge \mathsf{HW}(\mathbf{h}) \le \omega$$

$\underline{\mathsf{P}_1(\mathsf{sk}=(\rho,tr,\mathbf{s},\mathbf{e},\mathbf{t}_1,\mathbf{t}_0))}$

1: $\mathbf{A} := \mathsf{ExpandA}(\rho)$
2: $\mathbf{y} \xleftarrow{\$} \tilde{S}_{\gamma_1}^\ell$; $\mathbf{e}_w \xleftarrow{\$} S_\eta^k$
3: $\mathbf{w} := \mathbf{A}\mathbf{y}+\mathbf{e}_w \bmod q$
4: $(\mathbf{w}_1,\mathbf{w}_0) := \mathsf{DecomposeMod}_q(\mathbf{w},2\gamma_2)$
5: $(\mathbf{w}_1,\mathsf{st}=(\mathbf{w}_0,\mathsf{sk},\mathbf{y},\mathbf{e}_w))$

$\underline{\mathsf{P}_2(\mathsf{st},c)}$

1: $\mathbf{z} := c\mathbf{s}+\mathbf{y}$
2: $\bar{\mathbf{w}}_0 := \mathbf{w}_0 - c\mathbf{e}$
3: **if** $\|\mathbf{z}\|_\infty \ge \gamma_1-\beta$ **or** $\|\bar{\mathbf{w}}_0\|_\infty \ge \gamma_2-\beta$ **then return** $(\perp,\perp)$
4: **else**
5: $\quad \mathbf{h} := \mathsf{MakeHint}_q(-c\mathbf{t}_0, \mathbf{A}\mathbf{z}-c\mathbf{t}+c\mathbf{t}_0, 2\gamma_2)$
6: $\quad$ **if** $\|c\mathbf{t}_0\|_\infty \ge \gamma_2$ **or** $\mathsf{HW}(\mathbf{h}) > \omega$ **then** $\mathbf{h} = \perp$
7: $\quad$ **return** $(\mathbf{z},\mathbf{h})$

Correctness. We first prove the correctness of a variant of the scheme presented in Algorithm 1, which replaces Line 5 of P_2 with $\mathbf{h} := \mathsf{MakeHint}_q(-c\mathbf{t}_0+\mathbf{e}_w, \mathbf{A}\mathbf{z}-c\mathbf{t}+c\mathbf{t}_0, 2\gamma_2)$ and the first check of Line 6 with $\|c\mathbf{t}_0-\mathbf{e}_w\|_\infty \ge \gamma_2$, respectively. This variant is already MPC-friendly, as our proof in the next subsection implies that $\mathbf{e}_w$ is not sensitive once both checks in Line 3 of P_2 have passed, allowing us to release $\mathbf{e}_w$ in the clear.

We show that, conditioned on $\mathbf{z} \neq \perp$ and $\mathbf{h} \neq \perp$, the modified scheme is perfectly correct. Recall that Line 4 of Ver satisfies $\mathbf{w}' := \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d \equiv \mathbf{A}\mathbf{z} - c\mathbf{t} + c\mathbf{t}_0 \equiv \mathbf{w} - c\mathbf{e} + c\mathbf{t}_0 - \mathbf{e}_w \pmod q$. By Lemma 3, we have that

$$\begin{aligned}&\mathsf{UseHint}_q(\mathsf{MakeHint}_q(-c\mathbf{t}_0+\mathbf{e}_w,\mathbf{w}',2\gamma_2),\mathbf{w}',2\gamma_2)\\ &\quad= \mathsf{HighBits}_q(\mathbf{w}'-c\mathbf{t}_0+\mathbf{e}_w,2\gamma_2)\\ &\quad= \mathsf{HighBits}_q(\mathbf{w}-c\mathbf{e},2\gamma_2).\end{aligned}$$

For the verification condition to hold, it remains to be shown that $\mathsf{HighBits}_q(\mathbf{w}-c\mathbf{e},2\gamma_2) = \mathsf{HighBits}_q(\mathbf{w},2\gamma_2) = \mathbf{w}_1$. This is handled by the second rejection condition of Line 3 of Sign. By Lemma 5, if $\|\mathbf{w}_0-c\mathbf{e}\|_\infty < \gamma_2-\beta$ (as it is guaranteed by Sign outputting a passing transcript) and $\|c\mathbf{e}\|_\infty \le \beta$, then we have that $\mathsf{LowBits}_q(\mathbf{w}-c\mathbf{e},2\gamma_2) < \gamma_2-\beta$. Then Lemma 4 guarantees $\mathsf{HighBits}_q(\mathbf{w}-c\mathbf{e},2\gamma_2) = \mathsf{HighBits}_q(\mathbf{w},2\gamma_2) = \mathbf{w}_1$. The verification conditions on $\mathbf{z}$ and $\mathbf{h}$ are clearly satisfied due to the corresponding checks on Line 3 and Line 6 of P_2, respectively.

The expected number of iterations caused by Line 3 is identical to that of the original ML-DSA. Let $p_{\mathbf{z}}$ the probability that $\mathbf{z}$ passes the check, and $p_{\bar{\mathbf{w}}_0}$ the probability that $\bar{\mathbf{w}}_0$ passes the check conditioned on $\|\mathbf{z}\|_\infty < \gamma_1-\beta$. Following §3.4 of the Dilithium specification [4], the probability that the checks in Line 3 of P_2 succeed is

$$p_{\mathbf{z}} \cdot p_{\bar{\mathbf{w}}_0} \approx e^{-N\cdot(\beta k/\gamma_2 + \beta\ell/\gamma_1)}. \tag{1}$$

In Table 2, we provide the expected numbers of repetitions (denoted by $M_{\mathsf{rep}} = 1/p_{\mathsf{pass}}$) due to the checks in Line 3.

The probability that the checks in Line 6 of P_2 pass changes in theory due to the additional error $\mathbf{e}_w$. However, since we set the parameter such that $\|\mathbf{e}_w\|_\infty \le \eta \ll \|c\mathbf{t}_0\|_\infty$, the empirical probability that Line 6 fails is around 0.02, which remains almost unchanged compared to the ML-DSA scheme [4].

Optimization: skipping addition by $\mathbf{e}_w$ when computing hint. While the correctness analysis above assumes that MakeHint takes into account the overflow caused by $\mathbf{e}_w$, thanks to our parameter configuration, we can further optimize the construction by dropping $\mathbf{e}_w$ when computing a hint, as presented in Algorithm 1. In our empirical experiments, this optimization does not noticeably affect the probability of the hint growing beyond the size limit, and therefore this optimization does not sacrifice correctness in practice.

3.1 Defining Unforgeability

UF-CMA$_\perp$: Unforgeability under Chosen Message Attacks and Incomplete Signatures. We first discuss our extended version of the unforgeability notion tailored to our modified ML-DSA (or, more generally, signature schemes that sometimes output incomplete signatures), which we call *unforgeability under chosen message attack and incomplete signatures* (UF-CMA$_\perp$). Recall that the standard UF-CMA security game requires that no PPT adversary can produce a valid signature on a message that has never been queried to the signing oracle. While ML-DSA in its original form always outputs a valid signature after repeating the signing process several times, our modified version runs the signing process *only once*, and sometimes outputs an incomplete signature $(\mathbf{w}_1,c,\perp)$ whenever the norm check fails. We opted for this design choice to model an adversary who can potentially obtain an incomplete signature after a single signing attempt, and then leverage this information to adaptively choose messages to query the signing oracle. As the unforgeability game

Algorithm 2: $\mathsf{UF\text{-}CMA}_\perp$ **Experiment**

$\underline{\mathsf{Exp}^{\mathsf{cma}_\perp}_{\mathcal{A}}(1^\lambda)}$
1: $M := \varnothing$
2: $(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)$
3: $(\mathsf{m}^*,\sigma^*) \leftarrow \mathcal{A}^{O,|\mathsf{H}\rangle}(\mathsf{pk})$
4: **return** $\mathsf{Ver}(\mathsf{pk},\mathsf{m}^*,\sigma^*) = 1 \wedge \mathsf{m}^* \notin M$

$\underline{O(\mathsf{m})}$
1: $\mu := \mathsf{H}(tr,\mathsf{m})$;
2: $(\mathbf{w}_1,c,(\mathbf{z},\mathbf{h})) \leftarrow \mathsf{GetTrans}(\mu)$
3: **if** $\mathbf{z} \neq \perp$ **then**
4: $\quad M := M \cup \{\mathsf{m}\}$
5: **return** $(\mathbf{w}_1,c,(\mathbf{z},\mathbf{h}))$

$\underline{\mathsf{GetTrans}(\mu)}$
1: $(\mathbf{w}_1,\mathsf{st}) \leftarrow \mathsf{P}_1(\mathsf{sk})$
2: $c = \mathsf{H}(\mu,\mathbf{w}_1)$
3: $(\mathbf{z},\mathbf{h}) \leftarrow \mathsf{P}_2(\mathsf{st},c)$
4: **return** $(\mathbf{w}_1,c,(\mathbf{z},\mathbf{h}))$

already allows an adversary to query the signing oracle with arbitrary messages, this also naturally models the repeated signing process on the same message.

Now, note that variants of the security game can be defined based on when a forgery is considered trivial. In the standard UF-CMA notion, any forgery on a message m previously queried to the oracle is deemed trivial, regardless of whether the oracle returned a complete or incomplete signature. However, if the oracle only returns an incomplete signature on m, intuitively, it should be still difficult to produce a valid signature on m. To formalize this intuition, we introduce the stronger $\mathsf{UF\text{-}CMA}_\perp$ security, which allows an adversary to win the game by querying the signing oracle with m^*, obtaining an incomplete signature on m^*, and later outputting a valid signature on m^*. We define the following notion in the quantum random oracle model (QROM) [10].

Definition 2 ($\mathsf{UF\text{-}CMA}_\perp$/UF-NMA Security)**.** *A modified ML-DSA scheme is* $\mathsf{UF\text{-}CMA}_\perp$ *secure in the quantum random oracle model if for every polynomial-time adversary* $\mathcal{A}$ *that makes classical queries to the signing oracle* O *and quantum queries to the random oracle* H*, there exists a negligible function* ε *such that* $\Pr[\mathsf{Exp}^{\mathsf{cma}_\perp}_{\mathcal{A}}(1^\lambda) = 1] \leq \varepsilon(\lambda)$*, where* $\mathsf{Exp}^{\mathsf{cma}_\perp}_{\mathcal{A}}$ *is defined in Algorithm 2. Moreover, if* $\mathcal{A}$ *does not have access to the signing oracle* O*, then the scheme is* UF-NMA *secure.*

3.2 $\mathsf{UF\text{-}CMA}_\perp$ Security of Modified ML-DSA

We now present our main theorem regarding our modified ML-DSA scheme. We sketch the proof idea here and defer the formal proof to the full version [9].

Theorem 1. *The signature scheme presented in Algorithm 1 is* $\mathsf{UF\text{-}CMA}_\perp$ *secure in the (quantum) random oracle model, assuming the* UF-NMA *security of the original ML-DSA and under the* $\mathsf{MLWE}_{q,N,k,\ell,\tilde{S}^\ell_{\gamma_1},\eta}$ *and* $\mathsf{MLWE}_{q,N,k,\ell,S^\ell_{\gamma_1-\beta-1},\eta}$ *assumptions.*

Security of Fiat-Shamir with Aborts. Since ML-DSA is constructed by applying the Fiat-Shamir transform to the underlying three-pass identification scheme ID, we first recall standard security proofs for Fiat-Shamir signatures in the QROM (cf. [28, Theorem 9-10] [34, Theorem 3]). Instead of proving $\mathsf{UF\text{-}CMA}_\perp$ from scratch, we reduce the $\mathsf{UF\text{-}CMA}_\perp$ security to a weaker notion known as the UF-NMA *(unforgeability under no message attack)* security, which only considers adversaries that do not have access to the signing oracle. As the verification algorithm of our modified ML-DSA in Algorithm 1 is identical to that of the original ML-DSA, the UF-NMA security of our modified ML-DSA follows from that of the original ML-DSA [4]. The reduction from $\mathsf{UF\text{-}CMA}_\perp$ to UF-NMA boils down to the ability to simulate the signing oracle's reponses without knowing the secret key, which is typically shown by proving that the underlying ID is *strong computational honest-verifier zero-knowledge (scHVZK)* [28], defined below.

Definition 3 (Strong Computational Honest-Verifier Zero-Knowledge (scHVZK))**.** *An identification protocol* $(\mathsf{Gen},\mathsf{P},\mathsf{V})$ *satisfies the scHVZK property if there exists a polynomial-time simulator* Sim *such that for all probabilistic polynomial-time distinguisher* $\mathcal{A}$*, the following advantage is negligible in* λ*:*

$$\varepsilon_{\mathcal{A}} := \left| \Pr\left[\mathcal{A}(\mathsf{trs},\mathsf{sk}) = 1 : \begin{matrix}(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)\\ \mathsf{trs} \leftarrow \mathsf{Trans}(\mathsf{sk})\end{matrix}\right] - \Pr\left[\mathcal{A}(\mathsf{trs},\mathsf{sk}) = 1 : \begin{matrix}(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)\\ \mathsf{trs} \leftarrow \mathsf{Sim}()\end{matrix}\right] \right|$$

where pk *is an implicit argument to* Trans, Sim, $\mathcal{A}$ *above, and* $\mathsf{Trans}(\mathsf{sk})$ *denotes the following procedure:* $(w,\mathsf{st}) \leftarrow \mathsf{P}_1(\mathsf{sk})$*;* $c \xleftarrow{\$} C$*;* $z \leftarrow \mathsf{P}_2(\mathsf{st},c)$*; output* $\mathsf{trs} = (w,c,z)$.

Putting it all together. Once scHVZK is proved, we take advantage of the *adaptive reprogramming* lemma by [34] to reduce the $\mathsf{UF\text{-}CMA}_\perp$ security to the UF-NMA security of ML-DSA. As a result, we obtain Theorem 1. Here, to formally prove that incomplete signatures on queried m do not help the adversary to come up with a complete signature on m, we rely on the ability to simulate rejected transcripts *without knowing challenge c in advance*. This will essentially allow us to skip reprogramming the random oracle whenever a signing oracle outputs an incomplete signature.

Proving scHVZK. At the heart of Theorem 1 is the ability to simulate both accepting and rejected transcripts. We provide

a sketch of the simulation and proof here and defer the full proof with a concrete reduction loss to the full version [9].

Theorem 2. *The modified ML-DSA identification implicit in Algorithm 1 is strongly computational HVZK under the* $\mathsf{MLWE}_{q,N,k,\ell,\tilde{S}^{\ell}_{\gamma_1},\eta}$ *and* $\mathsf{MLWE}_{q,N,k,\ell,S^{\ell}_{\gamma_1-\beta-1},\eta}$ *assumptions. That is, there exists a polynomial-time simulator* Sim *such that for any scHVZK distinguisher* $\mathcal{A}$*, there exist adversaries breaking either* $\mathsf{MLWE}_{q,N,k,\ell,\tilde{S}^{\ell}_{\gamma_1},\eta}$ *or* $\mathsf{MLWE}_{q,N,k,\ell,S^{\ell}_{\gamma_1-\beta-1},\eta}$.

Proof Sketch. The first challenge is that the error we are adding to $\mathbf{w} := \mathbf{A}\mathbf{y} + \mathbf{e}_w$ is too small to show zero-knowledge for the interactive protocol with the first message as $\mathbf{w}$. This is because the verifier can compute $\mathbf{w} + c\mathbf{t} - \mathbf{A}\mathbf{z} = \mathbf{e}_w + c\mathbf{e}$, where the public key $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}$. If the first message was the full value $\mathbf{w}$, the error $\mathbf{e}_w$ would need to be the size of a flooding term, or an additional rejection sampling must be performed on $c\mathbf{e} + \mathbf{e}_w$ in order to show that the public key error is hidden. Thus, we cannot naively rely on the existing scHVZK proof [28, 44] that aims to simulate $(\mathbf{w}, c, \mathbf{z})$.

On the one hand, we have that the accepting transcripts for ML-DSA signatures are *perfectly* simulatable even *without* any error in $\mathbf{w}$ (i.e. $\mathbf{e}_w = \mathbf{0}$). On the other hand, we must show the simulatability of *rejected* transcripts $(\mathbf{w}_1, \mathbf{c}, \perp)$.

The detailed description of the simulator is presented in the full version [9]. While it is somewhat similar the generic Fiat-Shamir with Aborts (FSwA) simulator of [28], our simulator is more involved due to the optimizations unique to ML-DSA. This simulator takes in the public key $\mathbf{t}$,[6] and it is parametrized by $p_{\mathbf{z}}$, i.e., the probability that $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$. At the beginning, the simulator samples uniform challenge $c \in C$. The simulator has two modes: With probability $p_{\mathbf{z}}$ (resp. $1 - p_{\mathbf{z}}$), it enters Mode 1 (resp. Mode 2) to simulate the case where $\mathbf{z}$ passed the norm check (resp. where $\mathbf{z}$ failed the norm check). Within Mode 1, the simulator further enters one of the two modes as follows. The simulator in Mode 1 first samples a uniform random $\mathbf{w}' \in \mathcal{R}_q^k$, computes its lower bits $\mathbf{w}'_0$, and checks its norm as in the real prover. If $\mathbf{w}'_0$ fails the norm check, then it enters Mode 1-A; else, it enters Mode 1-B.

Mode 1-A: Simulating $(\mathbf{w}_1, c, \perp)$ for passing $\mathbf{z}$ and rejected $\bar{\mathbf{w}}_0$: In this mode, the simulator samples every element of $\mathbf{w}_1$ "almost" uniformly from $W_1 := \{0, 1, \ldots, (q-1)/(2\gamma_2) - 1\}$, i.e., the range of $\mathsf{Decompose}$. Note that $\mathbf{w}_1$ is not entirely uniform due to the edge case of $\mathsf{Decompose}$, slightly biasing towards 0. At a high-level, we can prove this case is computationally indistinguishable by the MLWE assumption, since $\mathbf{w}' = \mathbf{A}\mathbf{z} + \mathbf{e}_w - c\mathbf{t}$ is pseudorandom thanks to the added noise $\mathbf{e}_w$.

Mode 1-B: Simulating $(\mathbf{w}_1, c, (\mathbf{z}, \mathbf{h}))$ for passing $\mathbf{z}$ and passing $\bar{\mathbf{w}}_0$: For passing transcripts, we maintain the same perfect HVZK property as the original ML-DSA identification. In this mode, the simulator enters the following "loop mode" to freshly sample a passing transcript: sample a uniformly random $\mathbf{z}$ such that $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$, sample a uniform error $\mathbf{e}_w \in S_\eta^k$, compute $\mathbf{w}' = \mathbf{A}\mathbf{z} - c\mathbf{t} + \mathbf{e}_w$, until $\|\mathbf{w}'_0\|_\infty < \gamma_2 - \beta$. This loop ends in expectation after $1/p_{\bar{\mathbf{w}}_0}$ steps, which is a polynomial in λ.[7] As in the standard Fiat-Shamir with aborts proofs, $\mathbf{z}$ is perfectly simulated. We also obtain an identical distribution of $\mathbf{w}_1 = \mathsf{HighBits}_q(\mathbf{A}\mathbf{y} + \mathbf{e}_w, 2\gamma_2)$, since the simulator is able to reproduce *exactly* the same check as in the real prover. This is because $\mathbf{w}'$ computed by the simulator is identically distributed to $\mathbf{w}' = \mathbf{w} - c\mathbf{e}$ in the prover, since

$$\mathbf{A}\mathbf{z} - c\mathbf{t} + \mathbf{e}_w = \mathbf{A}\mathbf{y} + \mathbf{e}_w - c\mathbf{e} = \mathbf{w} - c\mathbf{e}.$$

By Lemma 4, as long as the check $\mathsf{LowBits}(\mathbf{w}', 2\gamma_2) < \gamma_2 - \beta$ passes and by setting $\beta = \tau\eta = \max_{c,\mathbf{e}} \|c\mathbf{e}\|_\infty$, we have $\mathsf{HighBits}_q(\mathbf{w}') = \mathsf{HighBits}_q(\mathbf{w}) = \mathbf{w}_1$, so the overall transcript is perfectly simulated.

Mode 2: Simulating $(\mathbf{w}_1, c, \perp)$ for rejected $\mathbf{z}$: In this mode, the simulator samples a uniform $\mathbf{w} \in \mathcal{R}_q^k$ and outputs its higher bits $\mathbf{w}_1$. Thanks to the additional noise $\mathbf{e}_w$, this mode computationally simulates a real transcript by the MLWE assumption. Note that the the distribution of real $\mathbf{w} = \mathbf{A}\mathbf{y} + \mathbf{e}_w$ is not exactly equivalent to the standard MLWE sample, because it is conditioned on $c\mathbf{s} + \mathbf{y}$ failing the norm check. However, by invoking the tight reduction recently presented by del Pino and Niot [44], we can reduce this variant of MLWE to standard MLWE assumptions with bounded uniform secrets.

4 Threshold ML-DSA Protocol

Now we describe our threshold signing protocol for ML-DSA, starting in this section with the online phase. We discuss the instantiation of the offline phase in Section 4.3. Our functionality that captures threshold ML-DSA signing, Functionality $\mathcal{F}_{\mathsf{T\text{-}MLDSA}}$, is presented in Appendix B.1. In essence, $\mathcal{F}_{\mathsf{T\text{-}MLDSA}}$ allows for two commands: $(\textsc{Gen})$, which enables the parties to obtain shares of a secret-key together with its respective public key, and $(\textsc{Sign}, \mathsf{m})$, which allows the parties to obtain the respective signature. $\mathcal{F}_{\mathsf{T\text{-}MLDSA}}$ runs our MPC-friendly description of ML-DSA signing from Appendix 3, since this is the one our protocols will instantiate.

To instantiate $\mathcal{F}_{\mathsf{T\text{-}MLDSA}}$, we rely on a preprocessing functionality that outputs shares of the "commitment message" $\mathbf{w}_1, \mathbf{w}_0$. We model this as Functionality $\mathcal{F}_{\mathsf{Prep}}$ below, which is instantiated in Section 4.3. Note that the functionality outputs the shares of $\mathbf{y}$, $\mathbf{w}_0$, and $\mathbf{w}_1$ over the arithmetic domain.

[6] We assume that all parties hold the full public key $\mathbf{t}$, since providing only $\mathbf{t}_1$ to the verifier is strictly a performance optimization.

[7] To realize a strict polynomial time simulator, we can bound the number of loops by $O(\lambda)$, leading to a negligible statistical loss.

Functionality 1: $\mathcal{F}_{\mathsf{Prep}}$

The functionality proceeds as follows:

1: If the adversary inputs abort, send abort to all parties. Otherwise, continue.
2: $\mathbf{y} \xleftarrow{\$} \tilde{S}^{\ell}_{\gamma_1}$ // $\tilde{S}_{\gamma_1} = S_{\gamma_1} \setminus \{x \in S_{\gamma_1} : \exists x_i \in x, x_i = -\gamma_1\}$
3: $\mathbf{e}_w \xleftarrow{\$} S^{k}_{\eta}$
4: $\mathbf{w} = \mathbf{A}\mathbf{y} + \mathbf{e}_w \bmod q$
5: $(\mathbf{w}_1, \mathbf{w}_0) = \mathsf{DecomposeMod}_q(\mathbf{w}, 2\gamma_2)$
6: Receive $3t$ shares from the adversary, complete $[\![\mathbf{w}_0]\!]$, $[\![\mathbf{w}_1]\!]$, and $[\![\mathbf{y}]\!]$ from these, and send shares to the honest parties.

In addition to $\mathcal{F}_{\mathsf{Prep}}$, the parties also need a functionality for *secure comparison*, which enables performing rejection sampling securely in MPC. This functionality, modeled as $\mathcal{F}_{\mathsf{LTC}}$ (which stands for *less than constant*) takes as input shares of a value $[\![x]\!]$, a public bound B, and outputs $[\![b]\!]$ where $b = (x < B)$. $\mathcal{F}_{\mathsf{LTC}}$ is a fundamental building block in many MPC applications, and it admits many different instantiations depending on a range of trade-offs. We discuss concrete instantiations $\mathcal{F}_{\mathsf{LTC}}$ in the full version [9].

4.1 Performing Rejection Sampling Securely

One of the core operations in ML-DSA signing is *rejection sampling*, which consists of comparing certain values against some given bounds, and restarting the signing process if any of the values fall out-of-bounds. We present the formal functionality in the full version [9], which takes in secret sharings, compares their secret values to the relevant bounds, and outputs either 1 (fail) or 0 (pass) to the parties.

Below, we present Protocol Π_{RejSamp}, which realizes this functionality. Overall, the parties call $\mathcal{F}_{\mathsf{LTC}}$ to perform the comparisons, and the problem is reduced to a functionality $\mathcal{F}_{\mathsf{BatchedOR}}$ that checks whether an array of shared bits contains at least one 1, which here indicates at least one sample was rejected. We instantiate this via Protocol $\Pi_{\mathsf{BatchedOR}}$, which works by taking κ random linear combinations, opening them, and verifying all of them are zero. Unfortunately, this approach on its own does not work since it leaks non-trivial information on which values failed the check (to see this, consider the extreme example where many linear combinations are open: it would eventually leak the vector of checks itself), and this leakage is not allowed by our signature functionality. In fact, in our case, hiding the position of out-of-bound coefficients in $\mathbf{z}$ is crucial especially because we reveal challenge c corresponding to a rejected signature. Combining this information together, it has been shown that a key recovery attack is possible [50]. To securely verify these κ "check bits" b^i shared over $\mathbb{F}_2$ efficiently with no leakage, the parties will first lift them to a different, large field $\mathbb{F}_p$ ($|\mathbb{F}_p| \geq 2^{\kappa}$) using *dabits* [47]. They then combine these bits by simply adding them together over $\mathbb{F}_p$ and finally multiply the sum by a random element $u \in \mathbb{F}_p$ to re-randomize. Finally, they open $y = (\sum_{i=1}^{\kappa} b^{(i)}) \cdot u$, and if $y \neq 0$, then we know $v^{(j)} = 1$ for at least one $j \in [\ell]$ with overwhelming probability and hence the rejection sampling failed.

Functionality 2: $\mathcal{F}_{\mathsf{BatchedOR}}$

Input: On receiving shares $[\![\mathbf{v}]\!]_2$ of a vector $\mathbf{v} \in \{0,1\}^{\ell}$ from the honest parties:

1: Reconstruct $\mathbf{v}$ alongside the corrupt parties' shares, and send the shares to the adversary.
2: If all entries of $\mathbf{v}$ are 0, set $b \leftarrow 0$.
3: Else, set $b \leftarrow 1$ to all parties.
4: Send bit b to the adversary.
5: Once the adversary replies with continue, output b to the parties.

Protocol 3: $\Pi_{\mathsf{BatchedOR}}$

Parameters: Prime p such that $p \geq 2^{\kappa}$.
Input: Secret-shared $[\![\mathbf{v}]\!]_2$ of length ℓ. Proceed as follows:

1: **for** i from 1 to κ **do** $(s_1^{(i)}, \ldots, s_{\ell}^{(i)})_{i=1}^{\kappa} \leftarrow \mathcal{F}_{\mathsf{Coin}}[\{0,1\}^{\ell}]$
2: **for** i from 1 to κ **do** compute $[\![b^{(i)}]\!]_2 \leftarrow \left(\bigoplus_{j=1}^{\ell} s_j^{(i)} \cdot [\![v_j]\!]_2\right)$
3: Call $\mathcal{F}_{\mathsf{dabits}}[1]$, κ times, to receive $([\![r^{(1)}]\!]_p; [\![r^{(1)}]\!]_2), \ldots, ([\![r^{(\kappa)}]\!]_p; [\![r^{(\kappa)}]\!]_2)$
4: Parties compute and open $c^{(i)} \leftarrow \mathcal{F}_{\mathsf{Open}}([\![b^{(i)}]\!]_2 \oplus [\![r^{(i)}]\!]_2)$ for $i \in [\kappa]$
5: Parties compute $[\![b^{(i)}]\!]_p \leftarrow c^{(i)} + (1 - 2 \cdot c^{(i)})[\![r^{(i)}]\!]_p$ for $i \in [\kappa]$
6: Parties compute $[\![b]\!]_p \leftarrow \sum_{i=1}^{\kappa} [\![b^{(i)}]\!]_p$
7: Parties call $[\![u]\!]_p \leftarrow \mathcal{F}_{\mathsf{Rand}}$
8: Parties compute $[\![y]\!]_p \leftarrow \mathcal{F}_{\mathsf{Mult}}([\![b]\!]_p, [\![u]\!]_p)$
9: Open $y \leftarrow \mathcal{F}_{\mathsf{Open}}([\![y]\!]_p)$
10: **if** $y \neq 0$ **then return** 1. // indicates $\exists j \in [\ell] : v_j = 1$.
11: **else return** 0.

The following is proved in the full version of the paper [9]. The opened vector $\mathbf{c}$ can be simulated since each component of α is sampled randomly. If all inputs of $\mathbf{v}$ are 0, then $\mathbf{c} = \mathbf{0}$ always. Otherwise, with overwhelming probability, at least one $b^{(i)} = 1$ and thus at least one $c_{i_0} = 1$.

Lemma 1. *$\Pi_{\mathsf{BatchedOR}}$ UC-realizes $\mathcal{F}_{\mathsf{BatchedOR}}$ in the $(\mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Rand}}, \mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{Open}})$-hybrid model.*

For Π_{RejSamp}, note that we actually check bounds on an *absolute value* $\|z\| < B$. Over integers this would correspond to $-B < z < B$, but since negative values are represented in $\mathbb{F}_q$ by integers in the range $[(q+1)/2, q)$, we have that $\|z\| < B$ iff $0 \leq z < B$ or $q - B < z < q$. We can shift this by $B - 1$ to equivalently get

$$\|z\| < B \iff \begin{cases} B - 1 \leq z + B - 1 < 2B - 1 \\ \textbf{or} \\ q - 1 < z + B - 1 < q + B - 1. \end{cases}$$

Since z is represented in $\mathbb{F}_q := \{0,1,\ldots,q-1\}$, the latter inequality wraps around and becomes $0 \le (z+B-1) \bmod q < B$, so the two inequalities together translate to $(z+B-1) \bmod q < 2B-1$, which is the strict-less-than operation that we will compute. We do so using the functionality $\mathcal{F}_{\mathsf{LTC}}$.

Protocol 4: $\Pi_{\mathsf{RejSamp}}([\![\mathbf{z}]\!], [\![\bar{\mathbf{w}}_0]\!])$

Inputs: Secret-shared $[\![\mathbf{z}]\!]$ (length $\ell \cdot N$) and $[\![\bar{\mathbf{w}}_0]\!]$ (length $k \cdot N$). Proceed as follows:

1: Define $k_x := \ell N$ and $k_y := kN$. Define $B_x := \gamma_1 - \beta$ and $B_y := \gamma_2 - \beta$.
2: **for** i from 1 to k_x **do**
3: $\quad [\![u_i]\!]_2 \leftarrow 1 - \mathcal{F}_{\mathsf{LTC}}[2B_x - 1]([\![\mathbf{z}_i + B_x - 1]\!])$ // $u_i = 0$ if $\mathbf{z}_i + B_x - 1 < 2B_x - 1$; $u_i = 1$ o.w.
4: **for** i from 1 to k_y **do**
5: $\quad [\![u'_i]\!]_2 \leftarrow 1 - \mathcal{F}_{\mathsf{LTC}}[2B_y - 1]([\![\bar{\mathbf{w}}_{0,i} + B_y - 1]\!])$
6: Concatenate $(u_i)_{i=1}^{k_x}$ and $(u'_i)_{i=1}^{k_y}$ as $\mathbf{v} \in \mathbb{F}_q^{k_x+k_y}$
7: Let $b \leftarrow \mathcal{F}_{\mathsf{BatchedOR}}([\![\mathbf{v}]\!]_2)$. // 1 means at least one check failed
8: **return** b // 1 means rejection sampling failed

The following is proved in the full version of the paper [9]. Simulation is straightforward, since everything is local, given the outputs of the sub-functionalities.

Lemma 2. Π_{RejSamp} *UC-realizes* $\mathcal{F}_{\mathsf{RejSamp}}$ *in the* $(\mathcal{F}_{\mathsf{LTC}}, \mathcal{F}_{\mathsf{BatchedOR}})$*-hybrid model.*

4.2 Threshold ML-DSA: Online Phase

Now we are ready to present our threshold ML-DSA signing protocol, assuming the functionality $\mathcal{F}_{\mathsf{RejSamp}}$, and also the preprocessing functionality $\mathcal{F}_{\mathsf{Prep}}$.

Protocol 5: $\Pi_{\mathsf{T\text{-}MLDSA}}$

Key generation: Upon receiving (GEN) as input:

1: All parties call $\rho \leftarrow \mathcal{F}_{\mathsf{Coin}}[\{0,1\}^{256}]$, $[\![\mathbf{s}]\!] \leftarrow \mathcal{F}_{\mathsf{Rand}}[S_\eta^\ell]$ and $[\![\mathbf{e}]\!] \leftarrow \mathcal{F}_{\mathsf{Rand}}[S_\eta^k]$
2: $\mathbf{A} := \mathsf{ExpandA}(\rho)$
3: Locally compute $[\![\mathbf{t}]\!] = \mathbf{A}[\![\mathbf{s}]\!] + [\![\mathbf{e}]\!]$
4: Open $\mathbf{t} \leftarrow \mathcal{F}_{\mathsf{Open}}([\![\mathbf{t}]\!])$
5: Parties locally compute $(\mathbf{t}_1, \mathbf{t}_0) = \mathsf{Power2Round}_q(\mathbf{t}, d)$ // $\mathbf{t}$ is public
6: Compute $tr := \mathsf{H}(\rho, \mathbf{t}_1)$
7: Let $\mathsf{pk} = (\rho, \mathbf{t}_1)$
8: Parties store $(\rho, tr, \mathbf{t}, \mathbf{t}_1, \mathbf{t}_0, [\![\mathbf{s}]\!], [\![\mathbf{e}]\!])$ and **return** pk.

Preprocessing: Upon receiving (PREP) as input, the parties call $([\![\mathbf{w}_0]\!], [\![\mathbf{w}_1]\!], [\![\mathbf{y}]\!]) \leftarrow \mathcal{F}_{\mathsf{Prep}}$ and store the output.

Sign: Upon receiving (SIGN, m) as input, proceed as follows:

1: Compute $\mu := \mathsf{H}(tr, \mathsf{m})$
2: Open $\mathbf{w}_1 \leftarrow \mathcal{F}_{\mathsf{Open}}([\![\mathbf{w}_1]\!])$, and locally compute $c := \mathsf{H}(\mu, \mathbf{w}_1)$
3: Locally compute $[\![\mathbf{z}]\!] = c \cdot [\![\mathbf{s}]\!] + [\![\mathbf{y}]\!]$ and $[\![\bar{\mathbf{w}}_0]\!] = [\![\mathbf{w}_0]\!] - c \cdot [\![\mathbf{e}]\!]$
4: $b \leftarrow \mathcal{F}_{\mathsf{RejSamp}}([\![\mathbf{z}]\!], [\![\bar{\mathbf{w}}_0]\!])$.
5: **if** $b = 1$ **then return** $(\mathbf{w}_1, c, \perp, \perp)$.
6: **else** open $\mathbf{z} \leftarrow \mathcal{F}_{\mathsf{Open}}([\![\mathbf{z}]\!])$ and compute hint $\mathbf{h} = \mathsf{MakeHint}_q(-c\mathbf{t}_0, \mathbf{Az} - c\mathbf{t} + c\mathbf{t}_0, 2\gamma_2)$.
7: **if** $\|c\mathbf{t}_0\|_\infty \ge \gamma_2$ or $\mathsf{HW}(\mathbf{h}) > \omega$ **then return** $(\mathbf{w}_1, c, \mathbf{z}, \perp)$
8: **else return** $(\mathbf{w}_1, c, \mathbf{z}, \mathbf{h})$.

The following is proved in the full version of the paper [9]. All opened values are public values in the Gen and Sign algorithms of ML-DSA, so security is straightforward.

Theorem 3. $\Pi_{\mathsf{T\text{-}MLDSA}}$ *UC-realizes* $\mathcal{F}_{\mathsf{T\text{-}MLDSA}}$ *in the* $(\mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Rand}}, \mathcal{F}_{\mathsf{Open}}, \mathcal{F}_{\mathsf{Prep}}, \mathcal{F}_{\mathsf{RejSamp}})$*-hybrid model.*

4.3 Threshold ML-DSA: Offline Phase

We now describe the instantiation of the $\mathcal{F}_{\mathsf{Prep}}$ functionality, which is part of the offline (*i.e.* message-independent phase).[8] Due to space constraints, we will only present Π_{Prep} for ML-DSA-65 and ML-DSA-87 in the main body; ML-DSA-44 requires a different instantiation due to the fact that δ in DecomposeMod for this case is not a power-of-two, unlike in the other cases. We present $\Pi_{\mathsf{Prep\text{-}44}}$ in Appendix 4.3.1.

For instantiating $\mathcal{F}_{\mathsf{Prep}}$ for ML-DSA-65 and ML-DSA-87, we will make use of a bit decomposition functionality, $\mathcal{F}_{\mathsf{BitDec}}$, that takes as input sharings $w \in \mathbb{F}_q$ and returns binary sharings of their bit-decomposition. We describe it as Functionality $\mathcal{F}_{\mathsf{BitDec}}$ in Section B.1 in the Appendix. This functionality appears in several applications of MPC where direct access to the bits is needed. We make use of the instantiation from [43]. See the Full Version [9] for a discussion on other approaches and their trade-offs.

Lines 3-9 in Π_{Prep} below capture our MPC instantiation of DecomposeMod for ML-DSA 65 and 87, where $\delta = (q-1)/(2\gamma_2)$ is a power-of-2. The following theorem is proved in the full version [9].

Theorem 4. Π_{Prep} *UC-realizes* $\mathcal{F}_{\mathsf{Prep}}$ *in the* $(\mathcal{F}_{\mathsf{Rand}}, \mathcal{F}_{\mathsf{BitDec}}, \mathcal{F}_{\mathsf{dabits}}, \mathcal{F}_{\mathsf{Open}})$*-hybrid model.*

Protocol 6: Π_{Prep}

Parameters The bit-length $m := \lceil \log q \rceil$. A power-of-2 modulus $\delta := \frac{q-1}{2\gamma_2}$ and $d = \log \delta$.

1: Call $[\![\mathbf{y}]\!] \leftarrow \mathcal{F}_{\mathsf{Rand}}[\tilde{S}_{\gamma_1}^\ell]$ and $[\![\mathbf{e}_w]\!] \leftarrow \mathcal{F}_{\mathsf{Rand}}[S_\eta^k]$.
2: Locally compute $[\![\mathbf{w}]\!] = \mathbf{A}[\![\mathbf{y}]\!] + [\![\mathbf{e}_w]\!]$
3: Locally compute $[\![\mathbf{s}]\!] \leftarrow -\delta \cdot [\![\mathbf{w}]\!] + (q-1)/2$
4: Call $([\![\mathbf{s}_1]\!]_2, \ldots, [\![\mathbf{s}_m]\!]_2) \leftarrow \mathcal{F}_{\mathsf{BitDec}}([\![\mathbf{s}]\!])$
5: Call $\mathcal{F}_{\mathsf{dabits}}[k]$, d times, to receive

[8] We remark that the offline phase of our end-to-end protocol is not restricted to Π_{Prep}: instantiating $\mathcal{F}_{\mathsf{LTC}}$ also requires some preprocessed data in the form of dabits and edabits.

$(\llbracket \mathbf{r}_1 \rrbracket; \llbracket \mathbf{r}_1 \rrbracket_2), \ldots, (\llbracket \mathbf{r}_d \rrbracket; \llbracket \mathbf{r}_d \rrbracket_2)$
6: Compute and open $\mathbf{c}_i \leftarrow \mathcal{F}_{\mathsf{Open}}(\llbracket \mathbf{s}_i \rrbracket_2 \oplus \llbracket \mathbf{r}_i \rrbracket_2)$ for $i \in [d]$
7: Locally compute $\llbracket \mathbf{s}_i \rrbracket \leftarrow \mathbf{c}_i + (1 - 2 \cdot \mathbf{c}_i) \llbracket \mathbf{r}_i \rrbracket$ for $i \in [d]$
8: Locally compute $\llbracket \mathbf{w}_1 \rrbracket \leftarrow \sum_{i=1}^{d} 2^{i-1} \cdot \llbracket \mathbf{s}_i \rrbracket$
9: Locally compute $\llbracket \mathbf{w}_0 \rrbracket \leftarrow \llbracket \mathbf{w} \rrbracket - (2\gamma_2) \cdot \llbracket \mathbf{w}_1 \rrbracket$
10: **return** $\llbracket \mathbf{w}_0 \rrbracket, \llbracket \mathbf{w}_1 \rrbracket, \llbracket \mathbf{y} \rrbracket$

4.3.1 ML-DSA-44 Offline Phase

The protocol Π_{Prep} from Section 4.3 only captures the case where the operation "$s \bmod^{+} \delta$" in DecomposeMod is equivalent to the extraction of $d = \log \delta$ LSBs. As $\delta = 44$ is not a power of two in ML-DSA-44, we need a separate protocol to handle this case. The essential building block is $\pi_{\mathsf{Mod\text{-}44}}$, which takes as input (a vector of) sharings $\llbracket r \rrbracket$ of $r \in \mathbb{Z}_q$ together with its bit-wise sharings over the binary domain, and outputs a sharing $\llbracket r \bmod \delta \rrbracket$. Intuitively, $\pi_{\mathsf{Mod\text{-}44}}$ exploits the fact that $\mathbb{Z}_q^*$ has a multiplicative (cyclic) subgroup $G = \langle g \rangle$ of order δ, as δ divides $q - 1$. Therefore, for $r \in \mathbb{Z}_q$, one can obtain $r \bmod \delta$ by computing the discrete logarithm of $g^r \bmod q$ with respect to the generator g. To extract the discrete logarithm, we define a polynomial $F(\mathrm{x}) = a_0 + a_1 \cdot \mathrm{x} + \cdots + a_\delta \cdot \mathrm{x}^\delta$, such that $F(g^i \bmod q) = i$ for $i \in [0, \delta - 1]$, and thus we have that $F(g^r \bmod q) = r \bmod \delta$. Thus, computing $\llbracket r \bmod \delta \rrbracket$ boils down to evaluating the public polynomial F at the point $g^r \bmod q$ in a multi-party fashion.

The procedure $\pi_{\mathsf{Mod\text{-}44}}$ is then used in $\Pi_{\mathsf{Prep\text{-}44}}$ to instantiate the preprocessing for ML-DSA-44. Both are formally presented below. The proof of the following theorem is deferred to the full version [9].

Theorem 5. *$\Pi_{\mathsf{Prep\text{-}44}}$ UC-realizes $\mathcal{F}_{\mathsf{Prep}}$ in the $(\mathcal{F}_{\mathsf{Rand}}, \mathcal{F}_{\mathsf{BitLTC}}, \mathcal{F}_{\mathsf{edabits}}, \mathcal{F}_{\mathsf{dabits}}, \mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{Open}})$-hybrid model.*

Procedure 7: $\pi_{\mathsf{Mod\text{-}44}}(\llbracket \mathbf{r} \rrbracket; \llbracket \mathbf{r}_1 \rrbracket_2, \ldots, \llbracket \mathbf{r}_m \rrbracket_2)$

Parameters:
- Generator g of order 44 subgroup $G \subset \mathbb{Z}_q^*$ and elements $h_1 = g^1, h_2 = g^2, h_3 = g^{2^2}, \ldots, h_m = g^{2^{m-1}}$.
- Polynomial $F(\mathrm{x}) = a_0 + a_1 \cdot \mathrm{x} + \cdots + a_{43} \cdot \mathrm{x}^{43}$, mapping elements of G injectively to $[44]$.

// Compute shares of g^r
1: Call $\mathcal{F}_{\mathsf{dabits}}[k]$, m times, to receive $(\llbracket \mathbf{s}_1 \rrbracket; \llbracket \mathbf{s}_1 \rrbracket_2), \ldots, (\llbracket \mathbf{s}_m \rrbracket; \llbracket \mathbf{s}_m \rrbracket_2)$
2: Compute and open $\mathbf{c}_i \leftarrow \mathcal{F}_{\mathsf{Open}}(\llbracket \mathbf{r}_i \rrbracket_2 \oplus \llbracket \mathbf{s}_i \rrbracket_2)$ for $i \in [m]$
3: Locally compute $\llbracket \mathbf{r}_i \rrbracket \leftarrow \mathbf{c}_i + (1 - 2 \cdot \mathbf{c}_i) \llbracket \mathbf{s}_i \rrbracket$ for $i \in [m]$
4: Locally compute $\llbracket \mathbf{u}_i \rrbracket \leftarrow \llbracket \mathbf{r}_i \rrbracket \cdot h_i + (1 - \llbracket \mathbf{r}_i \rrbracket)$
5: **for** i from 1 to $\log(m)$ **do**
6: **for** j from 1 to $\log(m)/2^i$ **do** // in parallel
7: Compute $\llbracket \mathbf{u}_j \rrbracket \leftarrow \mathcal{F}_{\mathsf{Mult}}(\llbracket \mathbf{u}_{2j-1} \rrbracket, \llbracket \mathbf{u}_{2j} \rrbracket)$
// Compute r mod 44 based on $F(\mathrm{x})$
8: Let $\llbracket \mathbf{x}_0 \rrbracket \leftarrow 1$ and $\llbracket \mathbf{x}_1 \rrbracket \leftarrow \llbracket \mathbf{u}_1 \rrbracket$
9: **for** i from 0 to $\lfloor \log(44) \rfloor$ **do**
10: **for** j from 0 to $2^i - 1$ **do**
11: Let $\llbracket \mathbf{x}_{2^{i+1}-j} \rrbracket \leftarrow \mathcal{F}_{\mathsf{Mult}}(\llbracket \mathbf{x}_{2^i} \rrbracket, \llbracket \mathbf{x}_{2^i - j} \rrbracket)$
12: Let $\llbracket \mathbf{r} \bmod 44 \rrbracket \leftarrow 0$
13: **for** i from 0 to 44 **do**
14: Compute $\llbracket \mathbf{r} \bmod 44 \rrbracket \leftarrow \llbracket \mathbf{r} \bmod 44 \rrbracket + a_i \cdot \llbracket \mathbf{x}_i \rrbracket$
15: **return** $\llbracket \mathbf{r} \bmod 44 \rrbracket$

Protocol 8: $\Pi_{\mathsf{Prep\text{-}44}}$

Parameters The bit-length $m := \lceil \log q \rceil$. A modulus $\delta = 44$.

1: Call $\llbracket \mathbf{y} \rrbracket \leftarrow \mathcal{F}_{\mathsf{Rand}}[\tilde{S}_{\gamma_1}^{\ell}]$ and $\llbracket \mathbf{e}_w \rrbracket \leftarrow \mathcal{F}_{\mathsf{Rand}}[S_\eta^k]$.
2: Locally compute $\llbracket \mathbf{w} \rrbracket = \mathbf{A} \llbracket \mathbf{y} \rrbracket + \llbracket \mathbf{e}_w \rrbracket$
3: Locally compute $\llbracket \mathbf{s} \rrbracket \leftarrow -\delta \cdot \llbracket \mathbf{w} \rrbracket + (q - 1)/2$
4: Call $\mathcal{F}_{\mathsf{edabits}}[m]$, k times, to receive $(\llbracket r^1 \rrbracket; \llbracket r_1^1 \rrbracket_2, \ldots, \llbracket r_m^1 \rrbracket_2), \ldots (\llbracket r^k \rrbracket; \llbracket r_1^k \rrbracket_2, \ldots, \llbracket r_m^k \rrbracket_2)$. Represent this as a vector of edabits $(\llbracket \mathbf{r} \rrbracket; \llbracket \mathbf{r}_1 \rrbracket_2, \ldots, \llbracket \mathbf{r}_m \rrbracket_2)$
5: Compute and open $\mathbf{a} \leftarrow \mathcal{F}_{\mathsf{Open}}(\llbracket \mathbf{s} \rrbracket + \llbracket \mathbf{r} \rrbracket)$
6: Compute $\llbracket \mathbf{b} \rrbracket_2 \leftarrow \mathcal{F}_{\mathsf{BitLTC}}[\mathbf{a}](\llbracket \mathbf{r}_1 \rrbracket_2, \ldots, \llbracket \mathbf{r}_m \rrbracket_2)$
7: Call $(\llbracket \mathbf{u} \rrbracket; \llbracket \mathbf{u} \rrbracket_2) \leftarrow \mathcal{F}_{\mathsf{dabits}}[k]$
8: Compute and open $\mathbf{d} \leftarrow \mathcal{F}_{\mathsf{Open}}(\llbracket \mathbf{b} \rrbracket_2 + \llbracket \mathbf{u} \rrbracket_2)$
9: Locally compute $\llbracket \mathbf{b} \rrbracket \leftarrow \mathbf{d} + (1 - 2 \cdot \mathbf{d}) \llbracket \mathbf{u} \rrbracket$
10: Call $\llbracket \mathbf{r} \bmod \delta \rrbracket \leftarrow \pi_{\mathsf{Mod\text{-}44}}(\llbracket \mathbf{r} \rrbracket; \llbracket \mathbf{r}_1 \rrbracket_2, \ldots, \llbracket \mathbf{r}_m \rrbracket_2)$
11: Locally compute $\llbracket \mathbf{w}_1 \rrbracket \leftarrow \mathbf{a} \bmod \delta + (1 - \llbracket \mathbf{b} \rrbracket) * (q \bmod \delta) - \llbracket \mathbf{r} \bmod \delta \rrbracket$
12: Locally compute $\llbracket \mathbf{w}_0 \rrbracket \leftarrow \llbracket \mathbf{w} \rrbracket - (2\gamma_2) \cdot \llbracket \mathbf{w}_1 \rrbracket$
13: **return** $\llbracket \mathbf{w}_0 \rrbracket, \llbracket \mathbf{w}_1 \rrbracket, \llbracket \mathbf{y} \rrbracket$

5 Performance and Experiments

We now discuss the concrete performance of our threshold signing protocol. For the online phase, we report both communication and runtimes derived from an implementation of our protocol. For the offline phase, we only report communication and round costs. For $\mathcal{F}_{\mathsf{Mult}}$ we first generate triples using DN07 [22], which costs $9.5n$ field elements in total, and then for the online phase two calls to $\mathcal{F}_{\mathsf{Open}}$ are required. We instantiate $\mathcal{F}_{\mathsf{Open}}$ in two possible ways: (1) via all-to-all reconstruction, which has a cost of $n \cdot (n-1)$ field elements in one round, and (2) via "multiple kings" as in [22], which costs $4n$ elements in two rounds. Recall that, due to limitations with Shamir secret-sharing, shares of binary secrets are actually defined over $\mathbb{F}_{2^k}$. We use a simple "packing optimization" that allows reconstruction of a batch of k secrets over $\mathbb{F}_2$, at the cost of one element in $\mathbb{F}_{2^k}$. We refer the reader to the full version [9] for details on these instantiations and the derivation of their costs. We set the statistical security parameter κ to 40.

5.1 Communication, Rounds, and Benchmarks

We present communication and rounds for both offline phase (i.e., PREP command in $\Pi_{\text{T-MLDSA}}$) and online phase (i.e., SIGN command in $\Pi_{\text{T-MLDSA}}$) of our protocol. We also provide runtime benchmarks for the online phase. See Table 3 and Table 4 for communication costs. We consider two variants of our protocol that differ on the method used for reconstruction: either all-to-all or king-based, as discussed above. SIGN involves: one arithmetic opening, one call to $\mathcal{F}_{\text{RejSamp}}$, and in case of *accept*, one more opening. Note that, for the communication-optimized version, the communication per party remains *constant*, whereas for the round-optimized variant this communication grows linearly with the number of parties. However, for a small number of parties, we observe that the round-optimized approach is likely the better option, since only a slight increase in the communication is required to halve the number of rounds. As the number of parties increases, the linear scaling in the round-optimized communication begins to diverge from the communication-optimized version. Applications with a large number of parties will likely favor the communication-optimized version, and we discuss below how to minimize the total number of expected rounds when generating a signature.

# of parties		3	7	15	31	63	Rounds
MLDSA-44	C	0.31	0.31	0.31	0.31	0.31	29
	R	0.15	0.47	1.1	2.36	4.87	16
MLDSA-65	C	0.43	0.43	0.43	0.43	0.43	29
	R	0.21	0.65	1.5	3.24	6.7	16
MLDSA-87	C	0.59	0.59	0.59	0.59	0.59	29
	R	0.29	0.88	2.06	4.42	9.14	16

Table 3: Online communication in MB per party for a successful SIGN command. For n parties, we set $t = (n-1)/2$. The C rows are communication-optimized and the R rows are round-optimized.

# of parties	3	7	15	31	63	Rounds
MLDSA-44	6	15.9	40.6	110.4	330.1	86 – 108
MLDSA-65	4.1	11.9	34.4	108.4	369.9	81 – 103
MLDSA-87	5.5	16	46.3	146	499	81 – 103

Table 4: Offline communication in MB per party for a successful SIGN command. For n parties, we set $t = (n-1)/2$. We also report round-count, given as a range since for the offline phase the number of rounds increases as the number of parties grow (this is due to the edabits protocol).

Full signature generation: parallel rejection sampling. As in the original ML-DSA scheme, the majority of invocations of the SIGN command will fail to output a complete signature. Table 2 gives the expected number of times the SIGN command must be run before a full signature is produced. Our variant of the ML-DSA signing algorithm has essentially the same failure rate, so parties can expect to invoke the SIGN command about four or five times before a signature is successfully produced.

For some applications, it may be acceptable to evaluate each SIGN command sequentially and only invoke the next SIGN command when the rejection sampling test fails. However, for most applications, this high variability in the number of rounds could cause unacceptable latency. To address this, we consider an approach[9] where the parties run several independent iterations of SIGN in parallel and open the first $\mathbf{z}$ value that passes the rejection sampling test. These parallel invocations of SIGN dramatically reduce the variance in the number of rounds while only slightly increasing the expected communication bandwidth of the protocol.

In Table 5, we give communication benchmarks for parallel invocations of the SIGN command. Note that we only perform the final opening of $\mathbf{z}$ on a *single* passing invocation; any other passing invocations must be discarded without opening the full signature. The table reports the expected performance of running multiple SIGN commands in parallel. Observe in Table 5 that evaluating more SIGN commands in parallel decreases the expected round-complexity but increases the expected communication.

	# parallel SIGN	1	2	4	8
	$\mathbb{E}$[# of Π_{RejSamp} iters]	5.09	2.78	1.69	1.20
Comm. opt.	$\mathbb{E}$[# of rounds]	149	82	50	35
	$\mathbb{E}$[per party comm.]	2.2	2.4	2.9	4.2
Round opt.	$\mathbb{E}$[# of rounds]	81	45	27	19
	$\mathbb{E}$[per party comm.]	7.6	8.5	10.3	14.5

Table 5: Expected number of rounds and communication in MB per party to generate an ML-DSA signature. We analyze this for both the round and communication-optimized versions of our online phase, in a setting with 15 parties generating ML-DSA-65 signatures.

Runtimes. We implemented[10] our circuits using Dalskov's C++ secure computation library (SCL) [21]. In Table 6, we give the runtimes for one round of rejection sampling ignoring network effects. Our benchmarks were run on a single thread of a machine with 32 GB of RAM with an Intel i7 chip running at 2.5 GHz. Additionally, we do not consider pipelining optimizations that would likely remove the computation from the overall wall-clock times. For example, in a WAN setting with a round-trip-time of 20 milliseconds, the overhead from network latency would be about 160 milliseconds for the round-optimized version and about 290 milliseconds for the communication-optimized version. In a pipelined im-

[9] Note that our approach is different from the one considered in [23, 44], in which a signature is generated only if all parties simultaneously succeed in local rejection sampling on *each share of* $\mathbf{z}$. In contrast, our protocol allows the parties to *jointly* perform rejection sampling on the shared $\mathbf{z}$, thereby preserving the expected number of restarts of the original ML-DSA scheme.

[10] https://doi.org/10.5281/zenodo.17888654

plementation, the network operations would run in parallel to the computation, so the overall wall-clock time would be heavily dominated by the network.

# of parties	3	7	15	31	63
ML-DSA-44	6.3	8.9	18.7	52.2	196.4
ML-DSA-65	8.7	12.5	25.7	70.4	261.2
ML-DSA-87	11.9	16.9	34.8	94.6	343.1

Table 6: Online compute times in ms ignoring network effects. All parties are evaluated in parallel.

5.2 Comparison to Related Works

In Table 7, we compare our protocol to similar related works, including the concurrent work by Borin et al. [11]. As noted in Section 1.2, while [11] supports dishonest majority and is more efficient in terms of communication and rounds for a small number of parties, ours achieves universal composability assuming honest majority and supports an arbitrary number of parties. To illustrate the cost of achieving arbitrary scalability and UC-security while remaining compatible with the ML-DSA verification process, we compare ours to (threshold) Raccoon [25], EKT [32], Ringtail [12], and Finally! [44]. These protocols are all based on the Raccoon signature scheme [27] and are proven secure under variants of game-based unforgeability notions [7] in the dishonest majority setting. While our communication and round counts are significantly higher, the signature and verification sizes remain the smallest thanks to the ML-DSA parameters.

Acknowledgments

We thank Anders Dalskov for answering our questions regarding the SCL library [21], Katharina Boudgoust for discussions on the security of ML-DSA, and the anonymous reviewers of USENIX Security 2026 for their valuable feedback and suggestions.

Disclaimer

Ethical Considerations

In this paper, we propose threshold ML-DSA signatures. To evaluate the ethical implications of our research project, we provide a stakeholder-based ethics analysis below.

Stakeholders

- Technical stakeholders: The authors; the cryptography community; and NIST MPTC organizers.
- Deployers and operators: Cloud providers; certificate authorities; financial institutions and custodians.
- Users and society: Users whose assets, identities, or communications rely on ML-DSA.

Impacts

- Ethical Principles: We articulate the ethical principles through Beneficence, Respect for Persons, Justice, and Respect for Law and Public Interest.
- Beneficence and Respect for Persons: As stated in the original submission, this paper focuses on the proposal of cryptographic protocols. Our protocols and experiments do not involve human subjects, user data, or any interaction with deployed systems. Justice, Respect for Law and Public Interest: By improving the security of signature schemes, this paper supports societal interests in resilience and trustworthy infrastructure.
- Harms: Stakeholders' expectations of security may be undermined if the scheme is used outside its proven threat model or if the security model is not clearly communicated. For instance, the scheme should not be considered secure once more than t key shares are corrupted. Moreover, the security of our protocol is not guaranteed if the corresponding implementation reuses ephemeral randomness (such as y, e_w and correlated randomness in the offline phase) across different signing sessions, or has non-constant time operations involving shared secrets.

Mitigations We mitigate these risks by (1) stating assumptions and threshold profiles, (2) following well-established security frameworks, i.e., game-based security analysis of our modified ML-DSA and simulation-based analysis of our threshold protocols in the UC framework, (3) clearly marking the implementation details in the code, emphasizing that ephemeral randomness is to be used only for a specific signing session and must not be reused or combined, and (4) warning

Scheme	ML-DSA	Security	Rounds (Online)	Comm. (Online)	$\lvert pk \rvert$	$\lvert sig \rvert$	t	n	N
This work (C)	✓	**UC-realize** $\mathcal{F}_{\text{T-MLDSA}}$	29	952	1.3	2.4	$< n/2$	$\leq \boldsymbol{N}$	**Any**
This work (R)	✓	**UC-realize** $\mathcal{F}_{\text{T-MLDSA}}$	16	1444	1.3	2.4	$< n/2$	$\leq \boldsymbol{N}$	**Any**
Borin et al. [11]	✓	Unforgeable	2	≤ 525	1.3	2.4	$< n$	$\leq N$	≤ 6
Finally! [44]	✗	Unforgeable	2	22	2.5	2.7	$< n$	$\leq N$	≤ 8
Ringtail [12]	✗	Unforgeable	1	11	4.5	13.4	$< n$	≤ 1024	Any
EKT [32]	✗	Unforgeable	1	14	5.5	11.1	$< n$	≤ 1024	Any
T-Raccoon [25]	✗	Unforgeable	3	40	3.8	12.4	$< n$	≤ 1024	Any

Table 7: Comparison with other lattice-based threshold signatures with 128-bit security. The column **ML-DSA** indicates whether the scheme produces signatures compatible with the standardized ML-DSA verification. Following Remark 1, N denotes the total number of parties participating in the DKG; n denotes the number of signing parties producing a signature; t denotes the corruption threshold, i.e., the security is guaranteed as long as up to t parties are malicious. Online communication is in KB per party, and public key and signature sizes are in KB. Per-party communication for [12, 25, 32] is derived by setting $n = 1024$, and for [44] by setting $n = 8$. The rounds and communication for ours and [11] are set to achieve a success probability > 0.5. We set $n = 7$ for our round-optimized protocol (R) to compute the communication. Per-party communication of our communication-optimized protocol (C) remains constant regardless of the number of parties n. We note that ours has an offline phase incurring varying communication costs depending on the number of parties (see Table 4).

that our proof-of-concept implementation is intended solely for research purposes and is not production-ready, as its side-channel resilience has not yet been evaluated.

Decision The anticipated societal benefits of enabling secure PQ threshold signatures outweigh the potential risks, which can be mitigated as stated above. Moreover, no personal data or human subjects are involved. Therefore, we conclude that conducting and publishing the research is ethically appropriate

Open Science

We have implemented and benchmarked the online phase of our threshold ML-DSA protocol. Our supplementary material, available at https://doi.org/10.5281/zenodo.17888654, includes all necessary code and instructions for reproducing our experiments.

References

[1] Shweta Agrawal, Damien Stehlé, and Anshu Yadav. Round-optimal lattice-based threshold signatures, revisited. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors, *ICALP 2022*, volume 229 of *LIPIcs*, pages 8:1–8:20. Schloss Dagstuhl, July 2022.

[2] Martin R. Albrecht, Russell W. F. Lai, Oleksandra Lapiha, and Ivy K. Y. Woo. Partial lattice trapdoors: How to split lattice trapdoors, literally. Cryptology ePrint Archive, Report 2025/367, 2025.

[3] Nabil Alkeilani Alkadri, Nico Döttling, and Sihang Pu. Practical lattice-based distributed signatures for a small number of signers. In Christina Pöpper and Lejla Batina, editors, *ACNS 2024, Part I*, volume 14583 of *LNCS*, pages 376–402. Springer, Cham, March 2024.

[4] Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: Algorithm specifications and supporting documentation (version 3.1). https://pq-crystals.org/dilithium/, 2021.

[5] Manuel Barbosa, Gilles Barthe, Christian Doczkal, Jelle Don, Serge Fehr, Benjamin Grégoire, Yu-Hsuan Huang, Andreas Hülsing, Yi Lee, and Xiaodi Wu. Fixing and mechanizing the security proof of Fiat-Shamir with aborts and Dilithium. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part V*, volume 14085 of *LNCS*, pages 358–389. Springer, Cham, August 2023.

[6] Mihir Bellare, Elizabeth C. Crites, Chelsea Komlo, Mary Maller, Stefano Tessaro, and Chenzhi Zhu. Better than advertised security for non-interactive threshold signatures. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 517–550. Springer, Cham, August 2022.

[7] Mihir Bellare, Stefano Tessaro, and Chenzhi Zhu. Stronger security for non-interactive threshold signatures: BLS and FROST. Cryptology ePrint Archive, Report 2022/833, 2022.

[8] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, September 2012.

[9] Alexander Bienstock, Leo de Castro, Daniel Escudero, Antigoni Polychroniadou, and Akira Takahashi. Quorus:

Efficient, scalable threshold ML-DSA signatures from MPC. Cryptology ePrint Archive, Report 2025/1163, 2025.

[10] Dan Boneh, Özgür Dagdelen, Marc Fischlin, Anja Lehmann, Christian Schaffner, and Mark Zhandry. Random oracles in a quantum world. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 41–69. Springer, Berlin, Heidelberg, December 2011.

[11] Giacomo Borin, Sofía Celi, Rafael del Pino, Thomas Espitau, Guilhem Niot, and Thomas Prest. Threshold signatures reloaded: ML-DSA and enhanced raccoon with identifiable aborts. Cryptology ePrint Archive, Paper 2025/1166, 2025.

[12] Cecilia Boschini, Darya Kaviani, Russell Lai, Giulio Malavolta, Akira Takahashi, and Mehdi Tibouchi. Ringtail: Practical Two-Round Threshold Signatures from Learning with Errors . In *2025 IEEE Symposium on Security and Privacy*, pages 149–164. IEEE Computer Society, May 2025.

[13] Cecilia Boschini, Akira Takahashi, and Mehdi Tibouchi. MuSig-L: Lattice-based multi-signature with single-round online phase. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 276–305. Springer, Cham, August 2022.

[14] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[15] Sofía Celi, Daniel Escudero, and Guilhem Niot. Share the MAYO: thresholdizing MAYO. In Ruben Niederhagen and Markku-Juhani O. Saarinen, editors, *PQCrypto 2025*, volume 15577 of *LNCS*, pages 165–198. Springer, 2025.

[16] Rutchathon Chairattana-Apirom, Stefano Tessaro, and Chenzhi Zhu. Partially non-interactive two-round lattice-based threshold signatures. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part IV*, volume 15487 of *LNCS*, pages 268–302. Springer, Singapore, December 2024.

[17] Yanbo Chen. DualMS: Efficient lattice-based two-round multi-signature with trapdoor-free simulation. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part V*, volume 14085 of *LNCS*, pages 716–747. Springer, Cham, August 2023.

[18] Jean-Sébastien Coron, François Gérard, Tancrède Lepoint, Matthias Trannoy, and Rina Zeitoun. Improved high-order masked generation of masking vector and rejection sampling in Dilithium. *IACR TCHES*, 2024(4):335–354, 2024.

[19] Jean-Sébastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun. Improved gadgets for the high-order masking of Dilithium. *IACR TCHES*, 2023(4):110–145, 2023.

[20] Elizabeth C. Crites, Chelsea Komlo, and Mary Maller. Fully adaptive Schnorr threshold signatures. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part I*, volume 14081 of *LNCS*, pages 678–709. Springer, Cham, August 2023.

[21] Anders Dalskov. SCL (Secure Computation Library)—utility library for prototyping MPC applications. https://github.com/anderspkd/secure-computation-library, 2022.

[22] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 572–590. Springer, Berlin, Heidelberg, August 2007.

[23] Ivan Damgård, Claudio Orlandi, Akira Takahashi, and Mehdi Tibouchi. Two-round n-out-of-n and multi-signatures and trapdoor commitment from lattices. *Journal of Cryptology*, 35(2):14, April 2022.

[24] Rafael del Pino, Thomas Espitau, Guilhem Niot, and Thomas Prest. Simple and efficient lattice threshold signatures with identifiable aborts. Cryptology ePrint Archive, Paper 2025/871, 2025.

[25] Rafaël Del Pino, Shuichi Katsumata, Mary Maller, Fabrice Mouhartem, Thomas Prest, and Markku-Juhani O. Saarinen. Threshold raccoon: Practical threshold signatures from standard lattice assumptions. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part II*, volume 14652 of *LNCS*, pages 219–248. Springer, Cham, May 2024.

[26] Rafael del Pino, Shuichi Katsumata, Guilhem Niot, Michael Reichle, and Kaoru Takemure. Unmasking TRaccoon: A lattice-based threshold signature with an efficient identifiable abort protocol. Cryptology ePrint Archive, Paper 2025/849, 2025.

[27] Rafaël del Pino, Shuichi Katsumata, Thomas Prest, and Mélissa Rossi. Raccoon: A masking-friendly signature proven in the probing model. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part I*, volume 14920 of *LNCS*, pages 409–444. Springer, Cham, August 2024.

[28] Julien Devevey, Pouria Fallahpour, Alain Passelègue, and Damien Stehlé. A detailed analysis of Fiat-Shamir with aborts. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part V*, volume 14085 of *LNCS*, pages 327–357. Springer, Cham, August 2023.

[29] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. Threshold ECDSA in three rounds. In *2024 IEEE Symposium on Security and Privacy*, pages 3053–3071. IEEE Computer Society Press, May 2024.

[30] Antonín Dufka, Semjon Kravtšenko, Peeter Laud, and Nikita Snetkov. Trilithium: Efficient and universally composable distributed ML-DSA signing. Cryptology ePrint Archive, Paper 2025/675, 2025.

[31] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 823–852. Springer, Cham, August 2020.

[32] Thomas Espitau, Shuichi Katsumata, and Kaoru Takemure. Two-round threshold signature from algebraic one-more learning with errors. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part VII*, volume 14926 of *LNCS*, pages 387–424. Springer, Cham, August 2024.

[33] Thomas Espitau, Guilhem Niot, and Thomas Prest. Flood and submerse: Distributed key generation and robust threshold signature from lattices. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part VII*, volume 14926 of *LNCS*, pages 425–458. Springer, Cham, August 2024.

[34] Alex B. Grilo, Kathrin Hövelmanns, Andreas Hülsing, and Christian Majenz. Tight adaptive reprogramming in the QROM. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part I*, volume 13090 of *LNCS*, pages 637–667. Springer, Cham, December 2021.

[35] Kamil Doruk Gür, Jonathan Katz, and Tjerand Silde. Two-round threshold lattice-based signatures from threshold homomorphic encryption. In Markku-Juhani Saarinen and Daniel Smith-Tone, editors, *Post-Quantum Cryptography - 15th International Workshop, PQCrypto 2024, Part II*, pages 266–300. Springer, Cham, June 2024.

[36] Shuichi Katsumata, Michael Reichle, and Kaoru Takemure. Adaptively secure 5 round threshold signatures from MLWE/MSIS and DL with rewinding. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part VII*, volume 14926 of *LNCS*, pages 459–491. Springer, Cham, August 2024.

[37] Eike Kiltz, Vadim Lyubashevsky, and Christian Schaffner. A concrete treatment of Fiat-Shamir signatures in the quantum random-oracle model. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 552–586. Springer, Cham, April / May 2018.

[38] Chelsea Komlo and Ian Goldberg. FROST: Flexible round-optimized Schnorr threshold signatures. In Orr Dunkelman, Michael J. Jacobson, Jr., and Colin O'Flynn, editors, *SAC 2020*, volume 12804 of *LNCS*, pages 34–65. Springer, Cham, October 2020.

[39] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *DCC*, 75(3):565–599, 2015.

[40] Yehuda Lindell. Simple three-round multiparty Schnorr signing with full simulatability. *CiC*, 1(1):25, 2024.

[41] Vadim Lyubashevsky. Lattice signatures without trapdoors. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 738–755. Springer, Berlin, Heidelberg, April 2012.

[42] Eleftheria Makri, Dragos Rotaru, Frederik Vercauteren, and Sameer Wagh. Rabbit: Efficient comparison for secure multi-party computation. In Nikita Borisov and Claudia Díaz, editors, *FC 2021, Part I*, volume 12674 of *LNCS*, pages 249–270. Springer, Berlin, Heidelberg, March 2021.

[43] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In Tatsuaki Okamoto and Xiaoyun Wang, editors, *PKC 2007*, volume 4450 of *LNCS*, pages 343–360. Springer, Berlin, Heidelberg, April 2007.

[44] Rafaël Del Pino and Guilhem Niot. Finally! A compact lattice-based threshold signature. In Tibor Jager and Jiaxin Pan, editors, *PKC 2025, Part III*, volume 15676 of *LNCS*, pages 169–199. Springer, Cham, May 2025.

[45] Federal Information Processing Standards Publication. FIPS 186-5: Digital Signature Standard (DSS). Federal Information Processing Standards Publication https://doi.org/10.6028/NIST.FIPS.186-5, 2023.

[46] Federal Information Processing Standards Publication. FIPS 204: Module-Lattice-Based Digital Signature Standard. Federal Information Processing Standards Publication https://doi.org/10.6028/NIST.FIPS.204, 2024.

[47] Dragos Rotaru and Tim Wood. MArBled circuits: Mixing arithmetic and Boolean circuits with active security.

In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *INDOCRYPT 2019*, volume 11898 of *LNCS*, pages 227–249. Springer, Cham, December 2019.

[48] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th FOCS*, pages 124–134. IEEE Computer Society Press, November 1994.

[49] Guofeng Tang, Bo Pang, Long Chen, and Zhenfeng Zhang. Efficient lattice-based threshold signatures with functional interchangeability. *IEEE Trans. Inf. Forensics Secur.*, 18:4173–4187, 2023.

[50] Yuanyuan Zhou, Weijia Wang, Yiteng Sun, and Yu Yu. Rejected signatures' challenges pose new challenges: Key recovery of CRYSTALS-dilithium via side-channel attacks. Cryptology ePrint Archive, Report 2025/214, 2025.

[51] Chenzhi Zhu and Stefano Tessaro. The algebraic one-more MISIS problem and applications to threshold signatures. Cryptology ePrint Archive, Paper 2025/436, 2025.

A Extended Preliminaries

A.1 Details of ML-DSA

In this section, we provide additional details on the ML-DSA scheme [4]. The list of complete parameters is provided in Table 2. In the description below, we replace some of the de-randomized operations in Gen and Sign in the spec by fully randomized ones. This modification does not impact interoperability with the standardized verification procedure.

Basic Operations. For a positive integer α and any $x \in \mathbb{Z}$, the centered modular reduction operation $x' := x \bmod^{\pm} \alpha$ maps x to a unique integer $-\lceil \alpha/2 \rceil < x' \leq \lfloor \alpha/2 \rfloor$ such that $x \equiv x' \bmod \alpha$. ML-DSA operates on a power-of-two cyclotomic ring $\mathcal{R} := \mathbb{Z}[x]/(x^N+1)$ and its quotient $\mathcal{R}_q := \mathcal{R}/q\mathcal{R}$. For a ring element $z = \sum_{i=0}^{N-1} z_i \cdot x^i \in \mathcal{R}_q$, the infinity norm of z is defined as $\|z\|_\infty := \max_i |z_i \bmod^{\pm} q|$. For a module element $\mathbf{z} = (z^{(1)}, \ldots, z^{(\ell)}) \in \mathcal{R}_q^\ell$, its infinity norm is defined analogously: $\|\mathbf{z}\|_\infty := \max_j \left\| z^{(j)} \right\|_\infty$. The Hamming weight $\mathsf{HW}(\mathbf{z})$ is defined as $||\mathbf{z}||_1 = \sum_j ||z^{(j)}||_1$.

We will write S_η to denote all elements $x \in \mathcal{R}_q$ such that $\|x\|_\infty \leq \eta$. We will write $\tilde{S}_\eta$ to denote the set $\{x \bmod^{\pm} 2\eta : x \in \mathcal{R}\}$. Note that $\tilde{S}_\eta$ does not contain any polynomials with $-\eta$ coefficients.

Hash Functions. ML-DSA uses SHAKE-256 H essentially in three ways and maps its output to appropriated codomains: (1) public key digest $tr = \mathsf{H}(\rho, \mathbf{t}_1) \in \{0,1\}^{256}$, (2) message digest $\mu = \mathsf{H}(tr, \mathsf{m}) \in \{0,1\}^{512}$, and (3) Fiat-Shamir challenge $c = \mathsf{H}(\mu, \mathbf{w}_1) \in C = \{c \in \mathcal{R} : ||c||_1 = \tau, ||c||_\infty = 1\}$. As the details of these mappings are not relevant to our protocol, we use the same notation H and refer the reader to [4, §5.3] for more details.

Security Levels & Common Parameters. The ML-DSA FIPS standard [46, Table 1, page 15] lists the various parameters for the scheme. In Table 1, we give the parameters that are unchanged for all security levels of ML-DSA. Note that $q := 2^{23} - (2^{13} - 1)$ just under 2^{23}, and elements in $\mathbb{Z}_q$ fit in 23 bits.

Supporting Functions. ML-DSA relies on the following supporting functions. We assume that if the function takes a module element $\mathbf{x} \in \mathcal{R}_q^k$ as input, it applies the operation coefficient-wise to each polynomial in $\mathbf{x}$. See [46, Algorithm 37]) for the full details:
$\mathbf{A} = \mathsf{ExpandA}(\rho)$: Expands a uniformly random 256-bit seed ρ into a larger matrix $\mathbf{A} \in \mathcal{R}_q^{k \times \ell}$.
$(r_1, r_0) = \mathsf{Power2Round}_q(r, d)$: Decomposes $r \in \mathbb{Z}_q$ into (r_1, r_0) such that $r = 2^d \cdot r_1 + r_0$
$(r_1, r_0) = \mathsf{DecomposeMod}(r, \alpha)$: [19] Decomposes $r \in \mathbb{Z}_q$ into unique (r_1, r_0) such that $r = \alpha \cdot r_1 + r_0$, where $r_1 \in [0, (q-1)/\alpha]$ and $r_0 \in [-\alpha/2 + 1, \alpha/2]$ if $r_1 \neq 0$. It is an MPC-friendly and equivalent variant of Decompose [46, Algorithm 36], and we use them interchangeably.
$r_1 = \mathsf{HighBits}_q(r, \alpha)$: Extracts the high bits r_1 of r by internally running $\mathsf{DecomposeMod}(r, \alpha)$.
$r_0 = \mathsf{LowBits}_q(r, \alpha)$: Extracts the low bits r_0 of r by internally running $\mathsf{DecomposeMod}(r, \alpha)$.
$h = \mathsf{MakeHint}_q(z, r, \alpha)$: Creates a hint bit h, indicating whether adding z to r changes the high bits of r.
$r_1 = \mathsf{UseHint}_q(h, r, \alpha)$: Uses the hint vector h to obtain the corrected high bits r_1 of r.

Computational Assumption. We recall the standard module LWE assumption, required by the security of ML-DSA. Note that the unforgeability of ML-DSA also relies on (SelfTarget)MSIS, which we omit here.

Definition 4 (Module Learning with Errors (MLWE))**.** *For $q \in \mathbb{N}$ and N a power of two, define $\mathcal{R}_q := \mathbb{Z}_q[x]/(x^N+1)$. For $k, \ell, \eta \in \mathbb{N}$, let $\mathcal{D}$ be some distribution over $\mathcal{R}_q^\ell$. The decision-MLWE problem $\mathsf{MLWE}_{q,N,k,\ell,\mathcal{D},\eta}$ is the task of distinguishing $(\mathbf{A}, \mathbf{A}\mathbf{y} + \mathbf{e})$ from $(\mathbf{A}, \mathbf{u})$ with non-negligible advantage, where $\mathbf{A}$ is uniform in $\mathcal{R}_q^{k \times \ell}$, $\mathbf{y} \leftarrow \mathcal{D}$ and $\mathbf{e}$ is sampled coefficient-wise uniformly from $[-\eta, \eta]$, and $\mathbf{u}$ is uniform in $\mathcal{R}_q^k$. If $\mathcal{D}$ is a uniform distribution over a set S, we write $\mathsf{MLWE}_{q,N,k,\ell,S,\eta}$.*

Secret Key & Public Key Distributions. The public matrix is $\mathbf{A} \in \mathcal{R}_q^{k \times \ell}$, which is obtained by expanding a random seed $\rho \in \{0,1\}^{256}$ via ExpandA function, internally calling SHAKE-128 (see §5.3 of [4] and Alg.32 of [46]). The security level in ML-DSA is adjusted by growing the dimensions k and ℓ. The public key for ML-DSA consists of ρ and the top bits of

$\mathbf{t} = \mathbf{As} + \mathbf{e} = \mathbf{t}_1 \cdot 2^d + \mathbf{t}_0$ where the coefficients of both $\mathbf{s}$ and $\mathbf{e}$ are uniformly sampled from $[-\eta, \eta]$. The actual public key is $\mathbf{t}_1 := \mathsf{Power2Round}_q(\mathbf{t}, d)$.

ML-DSA Signing & Rejection Sampling. The coefficients of the mask $\mathbf{y} \in \mathcal{R}_q^\ell$ are uniform in $[-\gamma_1 + 1, \gamma_1]$. The rejection sampling loop begins by sampling a mask $\mathbf{y}$, then computes $\mathbf{w} = \mathbf{Ay}$. Let $\mathbf{w}_1 := \mathsf{HighBits}_q(\mathbf{w}, 2\gamma_2)$. Let us denote $\delta = \frac{q-1}{2\gamma_2}$. Then the $\mathsf{HighBits}_q$ operation (see [46, Algorithm 37]) internally decomposes $\mathbf{w}$ into unique $(\mathbf{w}_1, \mathbf{w}_0)$ such that $\mathbf{w} = 2\gamma_2 \cdot \mathbf{w}_1 + \mathbf{w}_0$, where coefficients of $\mathbf{w}_1$ are in canonical representation modulo δ (i.e., $\mathbf{w}_1 \in \{0, 1, \ldots, \delta - 1\}^{kN}$) and coefficient of $\mathbf{w}_0$ are in centered representation modulo $2\gamma_2$ (i.e., $\mathbf{w}_0 \in [-\gamma_2 + 1, \gamma_2]^{kN}$ if $\mathbf{w}_1 \neq 0$ and $\mathbf{w}_0 \in [-\gamma_2, \gamma_2]^{kN}$ if $\mathbf{w}_1 = 0$), respectively. Note that this decomposition is carefully defined so that $2\gamma_2 \cdot \mathbf{w}_1 \neq q - 1$ and thus the distance (modulo q) between any two distinct $2\gamma_2 \cdot \mathbf{w}_1$ and $2\gamma_2 \cdot \mathbf{w}_1'$ is at least $2\gamma_2$. In concrete instantiations of ML-DSA, $\mathbf{w}_1$ is very small since δ is either 44 or 16.

Next, $\mathbf{w}_1$ is hashed to $c \in C \subset \mathcal{R}_q$ together with $\mu \in \{0,1\}^{512}$, where C consists of polynomials with coefficients in $\{-1, 0, 1\}$ with hamming weight τ, and μ is a hash of the message m to be signed and $tr = \mathsf{H}(\rho, \mathbf{t}_1)$. Observe that τ sets c to be quite sparse, which is highly restrictive for the verification norm. Define $\mathbf{z} := \mathbf{y} + c \cdot \mathbf{s}$, and define $\bar{\mathbf{w}}_0 := \mathsf{LowBits}_q(\mathbf{w}, 2\gamma_2) - c \cdot \mathbf{e}$. The rejection sampling consists of two checks: $||\mathbf{z}||_\infty < \gamma_1 - \beta$, and $||\bar{\mathbf{w}}_0||_\infty < \gamma_2 - \beta$. The latter check is for the error term, where γ_2 is nearly the modulus q.

Once both rejection sampling tests pass, a final hint $\mathbf{h}$ is computed, which is a series of bits marking when

$$\mathsf{HighBits}_q(\mathbf{w} - c \cdot \mathbf{e}, 2\gamma_2) \neq \mathsf{HighBits}_q(\mathbf{w} - c \cdot \mathbf{e} + c \cdot \mathbf{t}_0, 2\gamma_2).$$

If $||c \cdot \mathbf{t}_0||_\infty \geq \gamma_2$ or the hamming weight of $\mathbf{h}$ (denoted by $\mathsf{HW}(\mathbf{h})$) is beyond ω, rejection sampling restarts. The final signature is $\sigma = (c, \mathbf{z}, \mathbf{h})$.

ML-DSA Verification. The verifier takes the public key $\mathbf{t}_1$ and computes $\mathbf{w}' = \mathbf{Az} - c\mathbf{t}_1 \cdot 2^d$ and $\mathbf{w}_1' = \mathsf{HighBits}_q(\mathbf{w}', 2\gamma_2)$. For all locations where $\mathbf{h}$ is 1, the verification corrects $\mathbf{w}_1'$ by adding 1 if $\mathbf{w}_0' > 0$ or subtracting 1 if $\mathbf{w}_0' \leq 0$. Finally verification computes $c' = \mathsf{H}(\mu, \mathbf{w}_1')$, where $\mu = \mathsf{H}(\mathsf{H}(\rho, \mathbf{t}_1), \mathsf{m})$ and accepts if $c = c'$ and $||\mathbf{z}||_\infty < \gamma_1 - \beta$.

Technical Lemma. We recall the technical lemma from the ML-DSA specification [4]. Lemma 5 is a direct consequence of Lemma 2 and 3 of [4].

Lemma 3 (Lemma 1 of [4]). *Supposed that q and α are positive integers satisfying $q > 2\alpha$, $q \equiv 1 \pmod{\alpha}$ and α even. Let $\mathbf{r}$ and $\mathbf{z}$ be vectors of elements in $\mathcal{R}_q$ where $\|\mathbf{z}\|_\infty \leq \alpha/2$, and let $\mathbf{h}, \mathbf{h}'$ be vectors of bits. Then the $\mathsf{HighBits}_q, \mathsf{MakeHint}_q$, and $\mathsf{UseHint}_q$ algorithms satisfy the following properties:*

1. $\mathsf{UseHint}_q(\mathsf{MakeHint}_q(\mathbf{z}, \mathbf{r}, \alpha), \mathbf{r}, \alpha) = \mathsf{HighBits}_q(\mathbf{r} + \mathbf{z}, \alpha)$
2. *Let $\mathbf{v}_1 = \mathsf{UseHint}_q(\mathbf{h}, \mathbf{r}, \alpha)$. Then $\|\mathbf{r} - \mathbf{v}_1 \cdot \alpha\|_\infty \leq \alpha + 1$. Further, if the number of 1's in $\mathbf{h}$ is ω, then all except at most ω coefficients of $\mathbf{r} - \mathbf{v}_1 \cdot \alpha$ will have magnitude of at most $\alpha/2$ after centered reduction modulo q.*
3. *For any $\mathbf{h}, \mathbf{h}'$, if $\mathsf{UseHint}_q(\mathbf{h}, \mathbf{r}, \alpha) = \mathsf{UseHint}_q(\mathbf{h}', \mathbf{r}, \alpha)$, then $\mathbf{h} = \mathbf{h}'$.*

Lemma 4 (Lemma 2 of [4]). *Let $\|\mathbf{s}\|_\infty \leq \beta$. If $\left\|\mathsf{LowBits}_q(\mathbf{r} - \mathbf{s}, \alpha)\right\|_\infty < \alpha/2 - \beta$, then*

$$\mathsf{HighBits}_q(\mathbf{r} - \mathbf{s}, \alpha) = \mathsf{HighBits}_q(\mathbf{r}, \alpha)$$

Lemma 5 (Lemma 3 of [4], adapted). *Let $(\mathbf{r}_1, \mathbf{r}_0) = \mathsf{Decompose}_q(\mathbf{r}, \alpha)$, $(\mathbf{r}_1', \mathbf{r}_0') = \mathsf{Decompose}_q(\mathbf{r} - \mathbf{s}, \alpha)$, and $\|\mathbf{s}\|_\infty \leq \beta$. Then we have that*

$$\|\mathbf{r}_0 - \mathbf{s}\|_\infty < \alpha/2 - \beta \Leftrightarrow \left\|\mathbf{r}_0'\right\|_\infty < \alpha/2 - \beta.$$

B Supplementary Material for Section 4

B.1 Missing Functionalities

Functionality 3: $\mathcal{F}_{\text{T-MLDSA}}$

Setup (DKG): On receiving (GEN) from all parties:

$\rho \xleftarrow{\$} \{0,1\}^{256}$
$\mathbf{A} := \mathsf{ExpandA}(\rho)$
$(\mathbf{s}, \mathbf{e}) \xleftarrow{\$} S_\eta^\ell \times S_\eta^k$
$\mathbf{t} := \mathbf{As} + \mathbf{e} \bmod q$
$(\mathbf{t}_1, \mathbf{t}_0) := \mathsf{Power2Round}_q(\mathbf{t}, d)$
$tr := \mathsf{H}(\rho, \mathbf{t}_1)$
$(\mathsf{pk}, \mathsf{sk}) := ((\rho, \mathbf{t}_1), (\rho, tr, \mathbf{s}, \mathbf{e}, \mathbf{t}_1, \mathbf{t}_0))$
Internally store sk
Send $(\mathsf{pk}, tr, \mathbf{t}, \mathbf{t}_0)$ to the adversary and then output pk to all parties

Prep: Upon receiving (PREP) as input:

$\mathbf{y} \xleftarrow{\$} \tilde{S}_{\gamma_1}^\ell$ // $\tilde{S}_{\gamma_1} = S_{\gamma_1} \setminus \{x \in S_{\gamma_1} : \exists x_i \in x, x_i = -\gamma_1\}$
$\mathbf{e}_w \leftarrow S_\eta^k$
$\mathbf{w} = \mathbf{Ay} + \mathbf{e}_w \bmod q$
$(\mathbf{w}_1, \mathbf{w}_0) = \mathsf{DecomposeMod}_q(\mathbf{w}, 2\gamma_2)$
Internally store $(\mathbf{w}_0, \mathbf{w}_1, \mathbf{y})$

Sign: Upon receiving (SIGN, m) as input:

$\mu := \mathsf{H}(tr, \mathsf{m})$
$c = \mathsf{H}(\mu, \mathbf{w}_1)$
$\mathbf{z} = c\mathbf{s} + \mathbf{y}$
$\bar{\mathbf{w}}_0 = \mathbf{w}_0 - c\mathbf{e}$
if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\bar{\mathbf{w}}_0\|_\infty \geq \gamma_2 - \beta$ **then**
    $(\mathbf{z}, \mathbf{h}) = (\perp, \perp)$
else
    $\mathbf{h} = \mathsf{MakeHint}_q(-c\mathbf{t}_0, \mathbf{Az} - c\mathbf{t} + c\mathbf{t}_0, 2\gamma_2)$
    if $\|c\mathbf{t}_0\|_\infty \geq \gamma_2$ or $\mathsf{HW}(\mathbf{h}) > \omega$ **then**

10: $\mathbf{h} = \perp$
11: Send $(\mathbf{w}_1, c, \mathbf{z}, \mathbf{h})$ to the adversary and then output the same to all callers

Functionality 4: $\mathcal{F}_{\mathsf{LTC}}[B]$

On receiving shares $[\![x]\!]$ from the honest parties:

1: Reconstruct $x \leftarrow [\![x]\!]$ and the shares of the corrupt parties, and send the shares to the adversary.
2: let $b = (x < B)$
3: Receive t shares from the adversary, complete $[\![b]\!]_2$ from these and send shares to the honest parties.

Functionality 5: $\mathcal{F}_{\mathsf{BitDec}}$

On receiving shares of $[\![\mathbf{w}]\!]$ from the honest parties:

1: Reconstruct $\mathbf{w}$ and the shares of the corrupted parties, $\mathbf{w}^j$ for $j \in \mathcal{C}$.
2: Send $\{\mathbf{w}^j\}_j$ to the adversary.
3: **for** $i = 1, \ldots, m$ **do**
4: Receive from the adversary corrupted parties' shares $\mathbf{w}^j$, for $j \in \mathcal{C}$
5: Using $\{\mathbf{w}^j\}_j$ and $\mathbf{w}$, compute sharing $[\![\mathbf{w}]\!]_2$
6: Output to the parties their shares of $[\![\mathbf{w}_1]\!]_2, \ldots [\![\mathbf{w}_m]\!]_2$

Functionality 6: $\mathcal{F}_{\mathsf{RejSamp}}$

On receiving shares of $[\![\mathbf{z}]\!]$ and $[\![\bar{\mathbf{w}}_0]\!]$ from the honest parties:

1: Reconstruct $\mathbf{z} \leftarrow [\![\mathbf{z}]\!]$ and $\bar{\mathbf{w}}_0 \leftarrow [\![\bar{\mathbf{w}}_0]\!]$, and the shares of the corrupt parties, and send the latter to the adversary.
2: **if** $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ and $\|\bar{\mathbf{w}}_0\|_\infty < \gamma_2 - \beta - \eta$ **then** set $b \leftarrow 0$
3: **else** set $b \leftarrow 1$
4: Send bit b to the adversary.
5: Once the adversary replies with continue, output b to the parties.

Functionality 7: $\mathcal{F}_{\mathsf{BitLTC}}[B]$

The functionality takes in m binary shares $[\![x_1]\!]_2, \ldots, [\![x_m]\!]_2$ from the honest parties, and proceeds as follows:

1: Reconstruct $x_1, \ldots, x_m$ alongside the corrupt parties' shares, and send the shares to the adversary.
2: Compute $x := \sum_{i=1}^{m} x_i 2^{i-1}$.
3: Compute $b := (B < x)$
4: Receive from the adversary corrupted parties' shares b^j for $j \in \mathcal{C}$.
5: Sample $[\![b]\!]_2$ based on b and $\{b^j\}_j$ and send the parties their shares.